DIGITAL CONTROL OF THE CZOCHRALSKI GROWTH OF GALLIUM ARSENIDE
SYSTEM REFERENCE MANUAL
Valid for Czochralski Growth Controller Software Version 2.4

Arizona State University
Semiconductor Materials Research Laboratory
College of Engineering & Applied Sciences
Tempe, AZ 85287

January 4, 1988

Scientific Report, April 1, 1987 - December 31, 1987

ARPA Order No.: 9099
Contract No.: F49620-86-C-0012
Contract Effective Date: 10/1/85
Contract Expiration Date: 3/31/90

Program Manager: G. H. Schwuttke
(602) 965-2672
Contract Monitor: Gary Witt
(202) 767-4931

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Prepared for:
Defense Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

Air Force Office of Scientific Research
AFOSR/NE
Bolling AFB, DC 20332

Scientific Report

DIGITAL CONTROL OF THE CZOCHRALSKI GROWTH OF GALLIUM ARSENIDE

SYSTEM REFERENCE MANUAL

Valid for Czochralski Growth Controller Software Version 2.4

Sponsored by

Defense Advanced Research Projects Agency

G. H. Schwuttke
Principal Investigator
(602) 965-2672

Arizona State University
Semiconductor Materials Research Laboratory
College of Engineering & Applied Sciences
Tempe, Arizona 85287

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release, distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR- 88-0019
6a. NAME OF PERFORMING ORGANIZATION Arizona State University Semiconductor Materials Lab.	6b. OFFICE SYMBOL (If applicable) SPA
7a. NAME OF MONITORING ORGANIZATION Air Force Office of Scientific Research	
6c. ADDRESS (City, State and ZIP Code) Arizona State University Tempe, AZ 85287	7b. ADDRESS (City, State and ZIP Code) AFOSR/NE Bolling AFB, D.C. 20332
8a. NAME OF FUNDING/SPONSORING AFOSR/NE BLDG 410 BOLLING AFB, DC 20332-6448	8b. OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F49620-86-C-0012	
10. SOURCE OF FUNDING NOS.	

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
~~61102F~~ 61102F	~~2306~~ C1	C3	

11. TITLE (Include Security Classification)
Digital Control of the Czochralski Growth of Gallium Arsenide System Reference Manual

12. PERSONAL AUTHOR(S)
Karl Riedling

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Scientific	FROM 4/1/87 TO 12/31/87	88/01/04	412

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Digital Control,
GaAs
Reference Manual,

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report provides an updated and extended description (Version 2.4) of the structure and the operation of the controller software developed for ~~ASU's~~ digital Czochralski Growth Control System (CGCS) for compound semiconductors. This manual outdates all previous versions.

The Controller Software Reference Manual discusses the design considerations applied to digital LEC crystal growth control, gives a short overview of the growth controller computer hardware and operating system environment, describes the functions of the CGCS from an operator's point of view, and delineates the internal operations of the controller software by discussing the controller software and algorithms. Various appendices provide tables of controller software tasks, routines, and variables, file format information, and lists of system messages and error codes.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS ☐	Unclass

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Witt	(202) 767-4932	NE

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

ABSTRACT

This report provides an updated and extended description (Version 2.4) of the structure and the operation of the controller software developed for ASU's digital Czochralski Growth Control System (CGCS) for compound semiconductors. This manual outdates all previous versions.

The Controller Software Reference Manual discusses the design considerations applied to digital LEC crystal growth control, gives a short overview of the growth controller computer hardware and operating system environment, describes the functions of the CGCS from an operator's point of view, and delineates the internal operations of the controller software by discussing the controller software and algorithms. Various appendices provide tables of controller software tasks, routines, and variables, file format information, and lists of system messages and error codes.

Table of Contents

List of Illustrations

Summary

This manual constitutes a comprehensive documentation of process control, system, and auxiliary software developed by Arizona State University with the target of designing a digital controller system for the Liquid Encapsulated Czochralski (LEC) growth of gallium arsenide single crystals.

Digital crystal growth control was chosen because of its significant advantages over the standard analog approach:

* Better reproducibility of process parameters and control actions.

* A higher degree of flexibility with respect to operation procedures and process parameters.

* Powerful process automation.

* Expanded process data logging facilities.

The digital Czochralski Growth Control System (CGCS) is based on a microcomputer built around an Intel 8085 microprocessor. The system hardware consists of commercial OEM components; the microcomputer features 16 KBytes of Read Only Memory (ROM) and 56 KBytes of Random Access Memory (RAM), an Intel 8231 Numeric Processor, two industrial standard 8" single sided, single density flexible disk drives, and the Analog/Digital and Digital/Analog Converters and Input/Output (I/O) hardware which it requires to interface to the Czochralski puller. In addition to a console CRT terminal, a line printer and a multi-channel chart recorder are provided. The controller computer was designed as a multi-purpose unit which permits, in addition to the actual process control, to execute auxiliary programs for the maintenance of disks and disk files, and for the preparation and evaluation of growth runs. The operating system used is Intel's Real-Time Multitasking Executive iRMX-80; a special system environment, RXISIS-II, was developed for the execution of utility and support programs.

The CGCS is wired to monitor process data in parallel to the standard analog growth controller; its output can alternatively replace the analog controller's output. For reasons of simplicity, it uses part of the analog system's signal conditioning and output circuitry. In particular, it provides the analog motor speed and the heater power controllers with speed and power setpoints. The digital system can be operated in five modes each of which is an inclusive set of the preceding ones:

(1) Monitoring: The CGCS collects data from the puller which can be displayed and recorded, but it does not control the puller.

(2) Manual: The CGCS controls the growth process but allows only to enter setpoints for the primary process parameters (temperatures, motor speeds). No closed-loop diameter control is possible.

(3) Diameter: This mode includes closed-loop diameter control, based on the standard weighing method. Special algorithms compensate for the buoyancy effects caused by the encapsulation melt.

(4) Diameter/ASC: In addition to the above features, an anomaly compensation technique is used, which makes the diameter calculated by the CGCS more reliable.

(5) Automatic: A special algorithm permits to maintain the crystal-melt interface at a constant location within the heater, regardless of the amount of melt depleted due to crystal growth.

The CGCS software allows to dynamically access any parameter, including the parameters of controller loops, by direct operator commands. An arbitrary number of data locations can be identified with a symbolic name, and displayed, modified, and used for the decision-making process built into the CGCS. Parameters may be "ramped" within an arbitrary time from their current to their intended final values. Commands may be recorded on special disk files which may be edited and replayed as "Macro" commands during a later run; the sequence and timing of the recorded commands is exactly reproduced. These commands can be arbitrarily interspersed with new commands entered on the console; the resulting command sequence may be recorded again, which gives the system a learning ability. Macro command files may comprise any number of commands and can easily be invoked by name. A special feature permits to execute Macro commands conditionally, i.e., if and when a specified relation between an arbitrary system parameter and a constant value is reached. These properties of the CGCS allow to execute the crystal growth process essentially automatically, without the necessity of operator interactions. Crystals grown under automatic control exhibit improved uniformity of their electrical and crystallographic properties, compared to conventionally produced LEC crystals. Process yield in terms of single crystals and of usable wafers per crystal is distinctly superior to the yield of the standard analog technique.

Summary

Great emphasis was put on the design of the operator-machine interface: A specially formatted CRT console screen provides information about all data measured by the CGCS. Command entry is interactive, with as much flexibility as possible for the format of the commands. Several help menus and extensive command prompts guide the operator. The dialogue between the operator and the CGCS can be recorded either on disk, or on a line printer; each item is tagged with the time when it was issued. This permits, in conjunction with the data recording facilities of the CGCS, to trace the effects of a particular operation or event; the data taken during a run can be submitted to various process analysis and modelling approaches.

The Scope and Structure of This Documentation

This documentation comprises the entire software for the Czochralski Growth Control System, and for functions related to it. Although it also addresses the hardware configuration and operation as far as necessary, it should not be considered a hardware manual. Particularly in the sections detailing the actual process controller operations (chapter 5.3), readers proficient in FORTRAN may find it advantageous to have the program listings at hand (which are very extensively commented, too); frequently, references are made within this manual to the names of program variables or routines. It is, however, not necessary to study the source programs for using this documentation. With regard to the volume of this Reference Manual, information on Intel's operating systems ISIS-II and iRMX-80 for which detailed publications are commercially available is kept as concise as possible. These publications and a number of additional documentations which may supplement the material presented here are listed in Appendix 1.

This System Reference Manual comprises seven main chapters and fourteen appendices. After a short introduction in chapter 1 to the LEC process for compound semiconductor crystal growth, and the considerations applying to its automation, the hardware environment of the CGCS is discussed in chapter 2, where also CGCS-specific details about the configuration of the computer hardware are presented. Appendix 2 contains a procedure for the initial setup and test of the CGCS computer.

Chapter 3 discusses the operating system environment of the CGCS, starting with some views on the specific demands imposed on the software in a real-time environment, and delineating subsequently the operating system firmware in Read Only Memory (ROM), and the disk based operating system emulator RXISIS-II which provides favorable conditions for the execution of auxiliary and supporting software. (A number of file management utility programs for use under RXISIS-II which have been written by the author are presented in Appendix 6.)

The functional concepts and design of the CGCS are the topic discussed in chapter 4; this section provides also some instructions for the use of the CGCS, as far as required for understanding the operation of the software.

Chapter 5 consists of three sub-sections which cover the structure and two distinct parts of the CGCS software, respectively: Chapter 5.2 describes the interface to and the functions of a large number of system interface routines; section 5.3, the growth controller software proper, broken down into

an operator interface, and the process control functions. Although the system interface routines covered by chapter 5.2 are, indeed, "black boxes" for the higher-level parts of the CGCS, a thorough understanding of their operations was nevertheless considered essential for a complete comprehension of the high-level controller software. The presentation of the controller routines is augmented in Appendix 14 by a discussion of the dynamic response of the generic PID- (Proportional-Integral-Derivative) controller routine under various operation modes.

Chapter 6 discloses the procedure required for combining the program modules discussed in chapter 5 into an operational process control program, which is, due to the complexity of the CGCS, not trivial either.

The final chapter 7 concludes the survey of the CGCS software by discussing three auxiliary programs which support the operation of the CGCS and which can be executed on the CGCS computer under RXISIS-II (or on an Intellec Series II Development System under ISIS-II).

The appendices not mentioned so far contain summaries of information which should be accessed easily for reference purposes, e.g., memory maps, disk error codes, disk file formats, system messages, or routine and variable names.

CGCS Program Versions

This section describes the "evolution" of the documentation for the Czochralski Growth Control System by listing the features introduced with each release. Information on the CGCS software as given in this documentation is based upon version 2.4 of the Czochralski Growth Control System.

Version 1.3: (October 19, 1985)

(Version 1.3 was the first program release actually used for growing gallium arsenide crystals.)

Version 1.4: (December 5, 1985)

(1) INITIALIZE sets the diameter setpoint to the seed diameter. (This feature was discontinued from Version 2.1 on.)

(2) The Diameter evaluation routines check for zero seed lift speed and disable diameter calculation in this case.

(3) An automatic RESET is executed when required.

(4) The calculated Diameter is recorded in the Data file.

Version 1.5: (February 1986)

(1) RESET permits the entry of an initial value for the Crystal Weight and/or the Length Grown. (The effect of RESET on the Crystal Weight is a new feature of this release.)

(2) The length of the crystal stored by the buoyancy compensation part of the diameter calculation routine was increased from 37.5 millimeters to 75 millimeters. The thickness of one "slice" is approximately 0.5 mm; the maximum permitted seed travel speed exceeds 200 mm/h.

(3) The actual Diameter value is automatically copied to the Diameter setpoints when any diameter controlled mode is entered.

Version 1.6: (February 18, 1986)

(1) The Data Dump facility was newly introduced. Extra records are written to the Data file in case of an error detected by the Diameter Evaluation routines.

(2) The crystal diameter is evaluated with the actual growth rate rather than with the (actual) seed lift speed.

(3) The Diameter Evaluation routines were modified to recover automatically from Speed Overflow errors. (In previous versions, such errors disabled the diameter evaluation permanently; a RESET command was required to recover from this condition.)

Version 2.0: (April 11, 1986)

(1) The number of ramping channels was increased from 8 to 20.

(2) The maximum number of Conditional commands is 8 rather than 2. Conditional commands entered while already 8 Conditional commands are pending are ignored. (In earlier versions, a Conditional Macro command issued while already two Conditional commands were pending replaced the older one).

(3) A Selective CLEAR command was introduced which permits to remove only those Conditional Macro commands from the Conditional Command queue which pertain to a specified Variable.

(4) The PLOT feature was implemented, providing 8 analog channels for the output of arbitrary INTEGER*2 parameters, plus a set of pre-processed system parameters (Temperatures, Diameter error, Growth Rate, and Crucible Position error).

(5) 8 INTEGER*2 DUMMY locations were provided as a Macro command scratchpad.

(6) The CGCS can be put into a TEST mode. (Program patches (in ANACNT) were required in previous versions to execute run simulations.)

Version 2.1: (October 13, 1986)

(1) An erroneous algorithm in the Diameter Evaluation routine was corrected which resulted in a relative error of the calculated Growth Rate in the order of 10 percent.

(2) The buoyancy compensation routines were re-designed. In particular, a new interpolation algorithm was used for the determination of the crystal diameter at the top surface of the boric oxide encapsulant. A partial compensation of

the effects caused by melt recession at the end of the growth process was provided.

(3) Two new operation modes of the PID controller routine are available with release 2.1. They provide different approaches for a safe "anti-windup" function which improves the dynamic behavior of the controller in its output limited regime.

(4) The scaling of the Heater and Base Temperature output to the chart recorder was improved. A Variable-defined output range permits a flexible adaptation of the chart recorder output to various operating conditions.

(5) A timeout for the printer interface was activated. This feature prevents a defective or un-selected printer from suspending the operation of the system.

Version 2.2: (October 24, 1986)

(1) A new, more stable diameter interpolation algorithm replaces part of the procedures introduced with Version 2.1.

(2) The melt recession compensation algorithms were improved. A numeric parameter permits to adapt the Diameter Evaluation routines to arbitrary degrees of melt recession.

(3) The (square of the) crystal diameter stored in a table internal to the Diameter Evaluation routine is checked for excessive deviations with respect to its previous value, and adjusted accordingly if necessary.

(4) A check for a possible boric oxide encapsulant height overflow permits to run the CGCS safely with increased boric oxide charges.

(5) Conditional command checking is disabled for several seconds after a new (Conditional or unconditional) Macro command was started, in order to make sure that at least the first command of a Macro file can be executed in any case.

(6) An improved Macro command execution sequence guarantees the proper processing of Macro commands even in the case of transient disk errors.

(7) The generation of the Data disk file which was performed by two tasks in previous versions (one, collecting data in a buffer, and one, writing the buffer to disk) was concen-

trated in one single task. This measure provides the memory space required for the installation of the other software enhancements and reduces the probability of a temporary system deadlock due to a lack of pool memory, with the penalty of a possible minor record timing inaccuracy in the case of very short intervals between Data file records.

Version 2.3: (December 5, 1986)

(1) A periodic memory check was provided in this release, comprising the RAM resident main program code.

Version 2.4: (August 11, 1987)

(1) The algorithms for the diameter evaluation were changed to determine the growth rate from the current rather than the previous crystal diameter.

(2) A safety limit was imposed on the calculated diameter value used internally by the Diameter Evaluation routine, which protects the operation of the CGCS in case of severe transients imposed on the measured system parameters.

(3) A parameter THETA was newly introduced into the motor PID controllers which permits to set any operation mode between the feed-forward algorithm used up to now (THETA = 256) and a plain PID controller function (THETA = 0).

1. Introduction

The Czochralski Growth Control System Reference Manual endeavors to give a comprehensive description of the computer hardware and the software used in the autonomous digital growth controller for the LEC process for GaAs. This controller, specifically, the software which performs the process control operations, will be referred to as "Czochralski Growth Control System" (CGCS) throughout this manual.

The first section of this documentation will be devoted to an overview of the LEC process and its automated digital control. The second part deals with the computer hardware used, and its implementation. Next, we will discuss the operating system environment of the CGCS, and the implementation of various auxiliary and utility programs. The last main chapter, finally, details the software design of the CGCS proper, starting with the description of interface routines which are otherwise to be considered as "black boxes" within the controller software, and ending with the discussion of the high-level control routines. The configuration of the modules constituting the CGCS into the final operational software package, and the description of some support programs for the CGCS will conclude the System Reference Manual.

1.1 The LEC Growth Process For Compound Semiconductors

The Czochralski process is gaining increased importance not only for the growth of high purity silicon crystals but also for the large scale production of compound semiconductors like gallium arsenide. Although Czochralski grown GaAs crystals do not yet reach low dislocation densities comparable to those obtainable with the major competitor process, the Bridgeman technique, the Czochralski process offers, nevertheless, significant advantages over boat growth processes:

* The stoichiometry and the purity of Czochralski-grown crystals is superior to the properties of boat-grown ones. Semi-insulating substrates can be obtained with less or without chromium doping.

* The Czochralski process is better suited for a large scale production, and it is therefore cheaper.

A Czochralski puller (Fig. 1) consists essentially of a heated crucible made of quartz or boron nitride which contains the semiconductor melt. A small single crystal rod, the seed, is immersed into the melt and slowly lifted. The melt whose temperature is kept slightly above the semiconductor's melting point solidifies at the interface to the seed; with the proper temperature distribution and seed lift speed, a cylindrical single crystal can be grown whose crystallographic orientation is determined by the orientation of the seed. The crucible and the seed are rotated in opposite directions in order to minimize the influence of potential inhomogeneities of the temperature distribution inside the furnace. An inert atmosphere, usually argon, prevents the oxidation of the melt and of the crystal.

The growth of compound semiconductors like GaAs is impeded by the fact that these materials tend to dissociate at higher temperatures. The two components are bound together only loosely, and the one with the higher gas pressure (in our case, arsenic) tends to evaporate to a greater degree than the other (gallium), which results in intolerable deviations from stoichiometry and, in consequence, in bad electrical characteristics. While the miscellaneous variations of the Bridgeman process employ hermetically sealed quartz ampoules to prevent the loss of the volatile component, two approaches are used in the Czochralski process, either individually or combined: First, the pressure of the inert atmosphere inside the puller is increased to several hundred psis in order to counterbalance the arsenic vapor pressure, and, second, the semiconductor melt and the part of the crystal next to it are en-

capsulated in a vitreous melt of boric oxide (hence "Liquid Encapsulated Czochralski" (LEC) process).

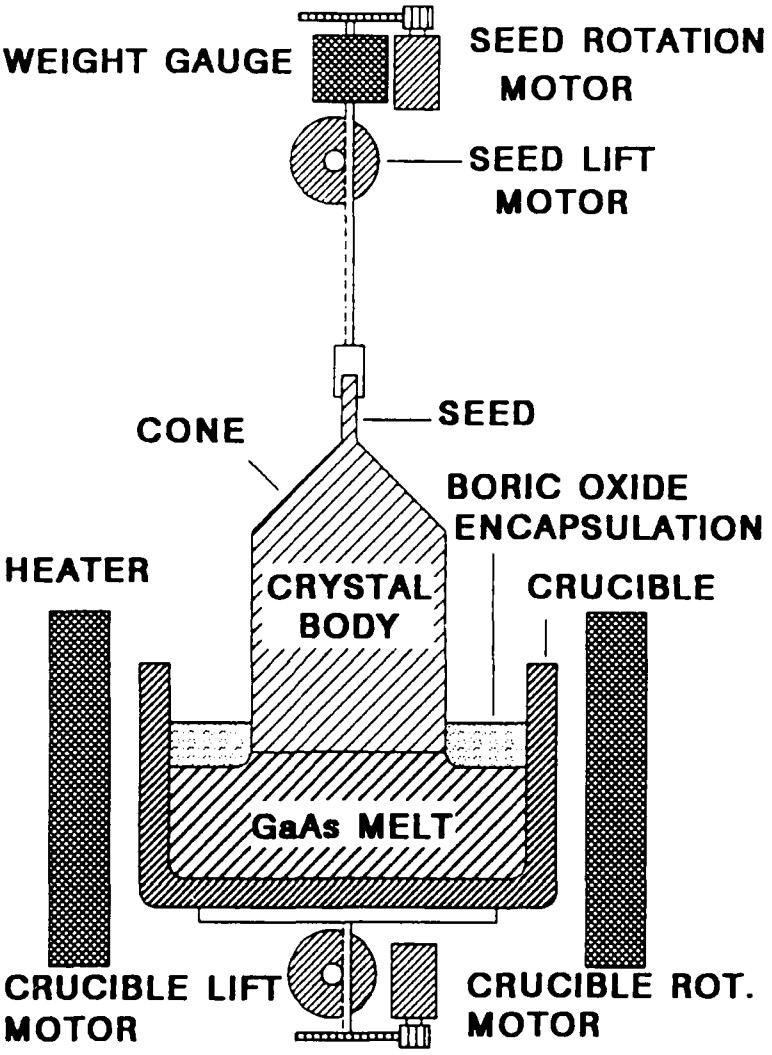


Fig. 1: A Czochralski puller for compound semiconductor crystal growth.

Technical applications of semiconductor single crystals require a defined, and preferably cylindrical, shape of the crystal ingots which have to be sliced into wafers with given dimensions. Semiconductor crystal growth implies, therefore, an efficient control of the diameter of the crystals grown.

Neither must the diameter drop below a minimum value (which would prohibit cutting a wafer with the specified diameter), nor should the diameter exceed its nominal value too much since the excess material is wasted as it must be ground away before the ingot is sliced into wafers. Conventional Czochralski pullers for compound semiconductors determine the diameter of the growing crystal from the increase of its weight per unit time which is obviously proportional to the crystal volume solidified per time. Taking a constant pull rate, i.e., a constant height of the incremental solid cylinder, for granted, this volume is proportional to the square of the crystal diameter. Diameter control can be effected by changing the temperature of the melt and/or the pull rate appropriately: The solidification of the molten semiconductor material generates heat which must be removed from the interface between the crystal and the melt in order to permit a continuous growth. The amount of heat which can be removed from the interface is, however, determined by the geometry of the furnace and of the crystal, and it is more or less constant. Increasing the temperature of the melt permits therefore less material to solidify, which results in a reduction of the crystal diameter if the pull rate is kept constant. On the other hand, an increase of the pull rate while the melt temperature is maintained has the consequence that the roughly constant volume of semiconductor material which can be solidified per unit time has to be stretched out to a longer and narrower cylinder, thus reducing the crystal diameter, and vice versa. (Compound semiconductors are, however, generally grown with temperature based diameter control since changes of the pull rate tend to deteriorate the material quality.)

A basic compound semiconductor puller features, therefore, the following elements (compare Fig. 1):

(1) A temperature controlled heater.

(2) Four speed controlled motors which are in charge of

   (a) the rotation of the crucible;

   (b) the rotation of the crystal;

   (c) the seed lifting motion; and

   (d) the lifting of the crucible which keeps the interface between the melt and the solid crystal at the same location within the heater in order to guarantee a constant temperature profile at the critical interface region.

(3) An electronic balance which permits to determine the crystal's weight and the weight increment; the latter signal can be used to control the heater temperature in order to maintain a defined crystal diameter.

Conventional compound semiconductor Czochralski pullers use analog electronic circuits to control the heater temperature and the motor speeds. Although this is an obvious approach (since all input and output parameters are inherently analog signals), there are several severe drawbacks associated with analog control circuitry:

(1) Analog controllers usually obtain their control parameters (e.g., the gain of a controller amplifier) from the setting of a potentiometer. It is not only difficult (and, frequently, impossible) to modify such parameters dynamically during a growth run although this might be desirable, it is also problematic to return to exactly the same settings which were used during earlier experiments once a parameter was changed.

(2) Despite of the fact that there are analog controllers on the market which feature a high degree of automation, the actual growth process is basically determined by the human operator. The high degree of human interaction, combined with the questionable repeatability of an analog system, makes it difficult to guarantee exactly reproducible growth conditions for subsequent growth runs.

(3) Crystal growth is, in fact, a very complex and not yet sufficiently understood process. A better understanding of the process which is the prerequisite for any process improvement can, however, be based only upon the thorough analysis of actual growth data. The logging of process data, particularly, of a greater number of data channels, is a very awkward procedure in an analog system; usually, crystal growers have to be content with in the order of three data channels logged on an analog chart recorder.

All these considerations favor the introduction of digital computer control for Czochralski crystal growth. A numerically based control permits not only absolute reproducibility of process parameters; it can much more readily be interfaced to automation approaches, and it permits, last but not least, the recording of growth data in a form suitable for later computer analysis.

1.2 A Digital Controller for GaAs Czochralski Growth

The basic target of the ASU project towards digital control of the Czochralski process for GaAs crystal growth was to replace the standard analog controller supplied by Cambridge Instruments, the company that built and delivered the puller proper, by a suitable computer-based controller. Since the entire setup is basically an experimental one, great emphasis had to be put on versatility and flexibility. Therefore, the approach shown in Fig. 2 was chosen:

The digital controller is connected in parallel to the standard analog system. Both systems monitor in parallel the output signals generated by the puller's sensors. Switches (actually, relays driven by the digital controller) permit to apply control signals to the puller either from the analog or from the digital controller. This allows, in conjunction with the proper software support, to switch control between both systems even during a growth run, which is particularly important during the setup and tuning of the digital controller. For reasons of simplicity, the digital system uses part of the signal conditioning circuitry and the motor controller and heater SCR circuits of the standard analog console. The digital system supplies, therefore, only motor speed and heater power setpoints; the standard analog controller's circuitry provides closed-loop motor speed and heater power control.

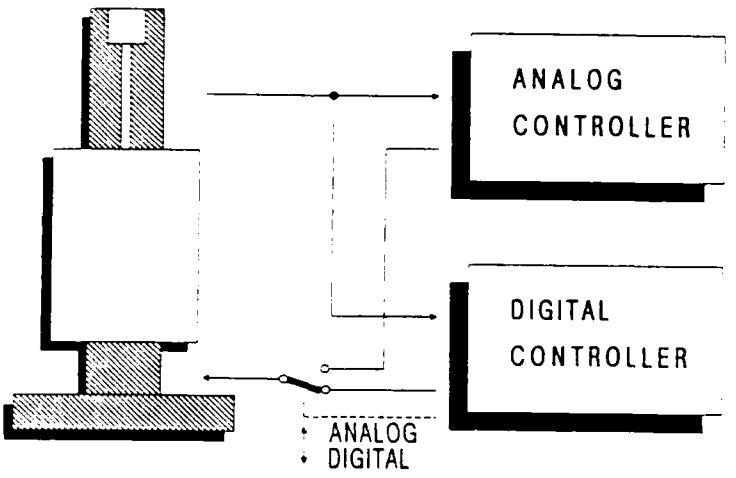


Fig. 2: Implementation of the digital Czochralski Growth Control System.

Furthermore, only those functions of the puller which directly affect the growth conditions are digitally controlled. Although the digital system is therefore not capable of running the puller entirely without the standard analog circuitry, this restriction to the most important operations permits to concentrate on features which are essential for the crystal growth, and facilitates the hardware and software implementation of the digital controller.

The following analog signal sources were chosen to be monitored by the digital controller, in parallel to the analog Cambridge console:

(1) Three thermocouples, measuring up to three heater zone temperatures. (Currently, only a single-zone heater is in use.)

(2) Four tachometers which are connected to the four motors for seed and crucible lift and rotation. (In contrast to the Cambridge Instruments terminology of "crystal" lift and rotation, we are using "seed" lift and rotation within this documentation and within the software, in order to avoid confusions of "crystal" and "crucible", particularly in abbreviations.)

(3) Up to three wattmeters which are connected to the puller's heater(s).

(4) The weight gauge monitoring the crystal weight.

(5) An analog differentiator circuit which generates a signal proportional to the first derivative of the crystal weight with regard to time. Determining the differential weight with an analog circuit rather than calculating it numerically from the plain weight was found advantageous because the crystal weight changes very slowly due to the slow growth of compound semiconductors. In order to allow to calculate with a reasonable resolution the differential weight from the plain weight in practical time intervals, the weight signal would have, therefore, required an extremely high analog-to-digital resolution, in excess of 20 bits. Suitable hardware is hardly commercially available, at least, for an affordable price. In contrast, a resolution of 14 bits is sufficient for all signals, including the plain weight, if analog weight differentiation is used.

(6) Two potentiometers which return voltages proportional to the current positions of the seed and the crucible, respectively.

(7) A thermocouple measuring the temperature at the bottom of the crucible ("base temperature").

(8) A pressure gauge sensing the pressure inside the puller's vessel.

(9) The "contact device" which is basically an ohmmeter circuit which monitors the resistance between the seed and the melt. This resistance drops from infinity to a certain value when the seed touches the (semiconducting) boric oxide encapsulation melt, and it drops further when contact between the seed and the actual semiconductor melt is established.

(10) Eight spare channels which can be used to record additional information (for example, the outputs of auxiliary thermocouples) together with growth data.

The signals which are supplied by the digital controller as replacements for the analog system's outputs are:

(1) Three heater SCR control voltages, anticipating a three-zone heater. (Currently, only one control voltage is used.)

(2) Four speed control voltages for the seed and crucible lift and rotation motors.

(3) Up to eight internal parameters can be submitted to a digital/analog conversion; the resulting eight analog signals can be recorded on a suitable multi-channel chart recorder.

In addition, digital signals are monitored by the digital controller and provided for the puller:

(1) Four motor direction signals: They are required, in addition to the (unipolar) speed control voltages, in order to determine the direction of motor motion (up or down, or clockwise or counterclockwise). The same control signals are also used within the standard analog controller; these signals generated by the analog circuitry are monitored by the digital system to provide complete status information.

(2) One master control signal: All control signal changeover relays are energized to select the digital system as a control signal source if this signal is present. Otherwise, the analog controller is in full charge of the puller. This is obviously an output-only signal of the computer system.

1.2 A Digital Controller for GaAs Czochralski Growth

The quasi-parallel operation of the analog and the digital controllers suggests a multi-step approach for the implementation of the computer-based system which is, indeed, supported by the digital controller software. Each of the following operation modes is upwardly compatible to the previous ones, providing all their functions plus some additional ones:

(1) Monitoring: The puller is still controlled by the analog system; the computer can be used to collect, display, and record measured data. This operation mode is evidently essential for establishing the proper operation of the data acquisition hard- and software, and it can be used to compare the actions of both controllers.

(2) Manual Growth: The control signals for the heater(s) and the four motors are generated by the digital system. Still, they result directly from temperature and speed setpoints, and no closed-loop diameter control is performed. The power applied to the heater(s) can be controlled in two ways: The system permits to provide three temperature setpoints, and one power limit value. The heater power output is determined by a temperature control loop while it is less than or equal to the limit value; it is set to the limit value if the temperature controller would request a greater heater power. The transition between both sub-modes is smooth and transparent to the user.

(3) Diameter Control: In this mode, the heater temperature is not only determined by its (manually entered) setpoint but also by a control loop which tries to keep the measured crystal diameter close to its corresponding setpoint. (For practical reasons, the "manual" temperature setpoint is corrected only slightly according to the diameter deviations, which results in a safer operation and gives improved control over the growth parameters.)

(4) Crucible Lift Control: The semiconductor melt in the crucible is gradually depleted while the crystal is grown. In order to maintain the solid-liquid interface at the same location within the heater, which is essential for reproducible crystal growth, the crucible has to be raised slowly during the growth run. This is done automatically in this operation mode, using a specially developed algorithm.

1.3 Crystal Growth Automation

A significant improvement of the current performance of the crystal growth process, in particular, of its yield, can only be expected if it is possible to grow crystals reproducibly, with repeatable properties. This implies, however, a higher degree of process automation in order to reduce the influence of the irregularities inevitably induced by human control actions. Evidently, a digital controller is much more suitable for automating a process than the conventional analog systems. (Although the Cambridge Instruments analog controller permits to control the crystal diameter automatically over large parts of the growth process, its total operation is far from automatic, and some very crucial operator actions are still required within the "automatic" growth phase.)

The digital Czochralski Growth Control System (CGCS) was, in general, designed to duplicate the existing analog controller. This is not true, however, for the approach chosen towards process automation. Our approach is not based upon a simple control of essentially one system parameter (namely, of the crystal diameter setpoint) but on the reproduction of all actions pertaining to the process. However, crystal growth is a highly complex operation which is strongly influenced by unforeseeable effects like random changes in the melt flow patterns in the crucible. It was, therefore, regarded an impossible task to automate an entire growth run by blindly repeating a fixed pattern of actions in a deterministic controller; we felt automation could only be achieved reasonably by splitting the process into small steps which are more promising targets for automatic control, and by the application of heuristic approaches. The system was, furthermore, designed to permit gradual improvements of such process steps, in order to optimize them more or less independently. The optimized steps can be joined together in a suitable way, being executed conditionally if required, to finally control an entire growth run.

The following features were therefore provided in the digital Czochralski Growth Control System in order to allow the optimization of the growth process:

(1) The system permits to modify interactively not only the actual growth data setpoints (for example, the diameter or the motor speed setpoints) but also any arbitrary internal system parameter ("Variable") which has an impact on the process. This applies specifically to the control loop parameters (e.g., to the gain of a control loop).

(2) The above changes can be made not only instantaneously but also slowly, by "ramping" a parameter linearly from its current to its intended final value within an arbitrary time. This approach prevents not only abrupt changes which are likely to upset a delicate process, it offers also a simple but efficient tool to automate process sequences. (For example, the cone between the seed and the crystal body can be grown by a ramp of the crystal diameter setpoint from the seed diameter to the intended crystal diameter over a time determined by the pull rate and the planned cone length.)

(3) Operator commands which affect the actual growth process can be optionally recorded on a disk file; the time at which a command was issued (relative to the start of command recording) is added as a tag to each command record. These "Macro" command files can be edited off-line, and invoked during a later growth run where they repeat exactly the recorded sequence of operator actions. Since prerecorded commands may be arbitrarily interspersed with commands entered by the operator during the run, and since the combined sequence of commands may be recorded again on a new disk file, the system achieves a "learning" ability. This command recording makes sense for self-contained process steps only (for example, for heating up the furnace, or for starting the growth proper), but it saves the operator a number of actions which frequently have to be done within a very limited time, and it prevents the inadvertent omission of important process steps.

(4) Further process automation can be achieved by the conditional execution of such Macro command files. A pre-recorded set of commands is started only if and when a system parameter which can be arbitrarily defined with the pertinent command incurs a certain numeric relation (e.g., greater than or equal) to a given constant. Such Conditional commands may also be issued from a Macro file; it is, therefore, possible to concatenate Macro files depending on the current status of the system. Even relatively complex process steps like seeding can thus be automated.

The current design of the Czochralski Growth Control System does effectively permit a fully automated growth. The task of the operator is reduced to supervising the process and interacting in the case of a malfunction (e.g., if the crystal "twins"). The current CGCS can not react to such events simply because it can not "see" them. Any attempt to include such features in an automated controller must therefore be based on the introduction of additional information, for example, of data supplied by suitable optical sensors.

2. The Hardware of the Czochralski Growth Control System

2.1 General Hardware Design

The digital Czochralski Growth Control system consists essentially of two parts which are linked together relatively loosely: One part, the "brains" of the system, is a suitable microcomputer, the other part is constituted by the hardware which interfaces the digital control computer to the essentially analog outside world. We will deal with both parts separately.

Microcomputer systems for industrial applications are usually designed exactly for the control task which they have to perform, i.e., with built-in software and a dedicated interface to the operator and to the process they have to control. Frequently, they feature only a very restricted set of function keys for operator input, and limited display facilities for the output of system status and data. We felt that such a system concept would hardly meet the requirements of an experimental system which was supposed to offer the following characteristics:

* Flexibility: The system software must be easy to modify, in order to adapt the system to varying demands, to introduce new features, and, last but not least, to correct programming errors.

* Versatility: The control computer should not only be able to control growth runs but also assist in the evaluation of measured data taken during crystal growth, and permit the preparation of experiments.

* Stand-alone operation: The growth controller computer should be used as a stand-alone unit, without requiring a host system for data transfer, evaluation, and maintenance.

* Interactive operation: The system should be run in an interactive mode, permitting a dialogue between the operator and the controller computer. This was regarded particularly important since the main target of the project was to learn about the dynamics of the crystal growth process, rather than producing crystals on a large scale according to pre-determined rules.

* Data display and logging facilities: As a consequence of the above considerations, it was regarded essential that the system should be able to display, evaluate, and record as many growth related parameters as possible.

All these demands cannot be fulfilled by a dedicated computer system with completely built-in software resident in ROM (Read Only Memory). It is not only an awkward procedure to modify ROM resident programs, particularly if frequent changes are required, it is even close to impossible to accommodate lengthy and frequently conflicting routines within the limited memory space available. Since it was necessary anyhow to provide mass storage devices for growth run data logging, we planned a generic disk-based microcomputer system which permits to load arbitrary programs from flexible disks. Command input to and data output from the control computer is handled by a standard CRT terminal which permits interactive operation and data display.

2.2 Computer Hardware

The hardware of the controller computer is based upon an Intel 8085 eight-bit microprocessor. This particular processor was chosen because of the vast experience we already had with it and because of the support software which was already available for it, which permitted to expect a fast system development. The experiences made with comparable applications showed that the processor's performance is sufficient if a system is well designed. The 8085 is able to address a 64 KByte memory space (plus 256 Input/Output (I/O) ports); with regard to the desired flexibility and versatility, as much of this memory space as possible was to consist of read-write memory (RAM - Random Access Memory). Only the absolute minimum of ROM which is indispensable for the operation of a computer was provided; the ROM resident code has, essentially, to control the loading of the actual application software from disk. The memory components available suggested, in addition, a memory bank switching approach which further reduces the amount of memory space consumed by ROM: The total ROM area of 16 KBytes is subdivided into two banks of equal size which can be activated alternately and which consume, therefore, only 8 KBytes of address space. One bank holds confidence test routines and a Monitor which are only needed for starting and/or debugging the system; the other bank is reserved for permanently required operating system routines. Therefore, 56 KBytes are available for RAM within the 64 KBytes address space; Fig. 3 shows a memory map of the controller computer.

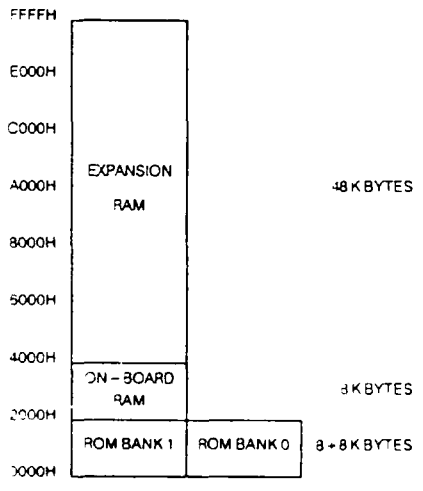


Fig. 3: Hardware memory map of the CGCS computer.

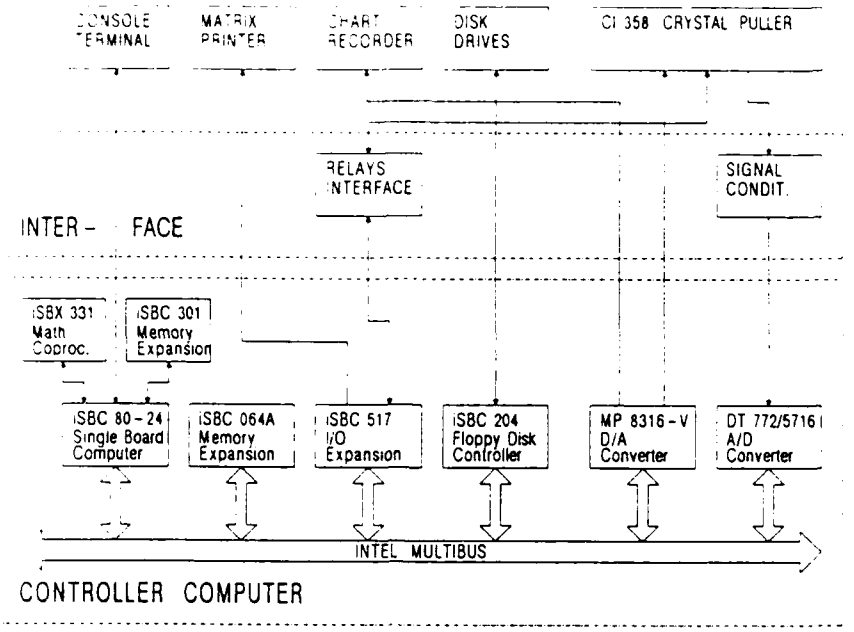


Fig. 4: Block diagram of the CGCS computer.

The controller computer is built of commercial OEM (Original Equipment Manufacturer) components most of which are supplied by Intel Corporation; these boards are interconnected via Intel's Multibus. The system configuration is shown in Fig. 4: An Intel iSBC 80-24 Single Board Computer board holds the 8085 CPU, the 2x8 KBytes of ROM, 8 KBytes of high-speed RAM, and an Intel 8231 Arithmetic Processing Unit (APU) on an iSBX 331 expansion board which permits to increase the throughput, particularly of data output to the system console. Two expansion boards, an iSBC 517 I/O, and an iSBC 064A Memory Expansion Board, provide additional I/O lines and the remaining 48 KBytes of RAM, respectively. (An iSBC 028A board was used in the ASU system instead of the iSBC 064A board due to availability reasons. Both boards are interchangeable in the controller computer; in either case, only 56 KBytes of the 64 KBytes RAM on the iSBC 064A, or of the 128 KBytes RAM on the iSBC 028A are used.) An iSBC 204 Floppy Disk Controller board constitutes the interface to the mass storage which consists of two (industrial standard) 8" single side, single density flexible disk drives with a storage capacity of 250 KBytes each.

A standard "dumb" CRT terminal serving as an operator console is connected to the iSBC 80-24 Single Board Computer via an RS-232 serial interface. A similar serial interface on the iSBC 517 I/O Expansion Board connects to a printer whose main task is providing a hard copy of the dialogue between the operator and the Czochralski Growth Control System.

The controller computer has to monitor and generate a number of analog and digital signals which were listed in chapter 1.2 of this documentation. The interface to the analog signals consists of one Analog-to-Digital (A/D) and one Digital-to-Analog (D/A) Converter board. Both boards are interconnected to the microcomputer proper via the Multibus system bus; data is read from and written to them via I/O port accesses.

The A/D Converter is a Data Translation DT772/5716-32DI-B-PGH board which features 32 differential input channels with a sensitivity of ±10 V (which may be increased by a factor of up to 8 under software control). The voltage of the (software selectable) input channel is converted into a 16 bit integer value by the board, corresponding to a resolution of 1/65,536; this data is read and eventually processed by the computer. A bank of isolation amplifiers between the signal sources and the A/D converter prevents ground loops which might induce noise and provides the necessary pre-amplification of low-level signals like the outputs of thermocouples.

The analog control voltages for the puller are output by a Burr-Brown MP8316-V D/A Converter board. This board features 16 channels with an output voltage swing of 0 ... 10 V; its resolution is 12 bit (1/4,096). Eight of the 16 output channels are reserved for the interconnection to an analog chart recorder for on-line data output.

Digital I/O of the motor direction information and of the controller selection is performed via a series of digital I/O ports on the iSBC 517 I/O Expansion Board. These signals are buffered and pre-processed by a simple external digital circuit. Relays constitute the actual input and output interface to the puller, permitting absolute isolation between the puller's circuitry and the computer.

2.3 Hardware Setup

2.3.1 iSBC 80-24 Single Board Computer

A detailed description of the operation of the iSBC 80-24 board is contained in Intel's iSBC 80-24 Single Board Computer Hardware Reference Manual (order No 142648-001). In general, the default settings listed there, and the hardware modifications specified in Intel's iRMX-80 User's Guide (order No 9800522-05) apply. If the following specifications contradict the data supplied by Intel, however, the information given in this documentation is valid.

The jumpers listed below have to be removed from the board (the numbers in parentheses refer to the sheet of the Intel schematics and to the location on this sheet):

E83 - E84 (9-B4)
E84 - E85 (9-B4)
E100 - E118 (9-C4)
E101 - E106 (9-C4)
E119 - E120 (9-C4)
E148 - E149 (4-C7)

The following jumpers have to be added:

E141 - E144 (3-B7): This jumper suppresses CPU wait states if ROM is accessed. Even 2732A EPROMs with 250 ns access time were found to be fast enough to be read without intervening wait states.
E102 - E91 (9-B4): Connects Multimodule interrupt MINTR1 (J6) to IR0.
E101 - E118 (9-C4): Connects output of Timer 0 to IR1.
E100 - E106 (9-C4): Connects MULTIBUS INT2/ to IR2.
E99 - E105 (9-C4): Connects MULTIBUS INT3/ to IR3.
E98 - E109 (9-C4): Connects MULTIBUS INT4/ to IR4.
E97 - E110 (9-C4): Connects MULTIBUS INT5/ to IR5.
E96 - E117 (9-B4): Connects 80-24 USART RxRDY to IR6.
E83 - E115 (9-B4): Connects 80-24 USART TxRDY via OR gate (U34-9) to IR7.
E84 - E112 (9-C4): Connects MULTIBUS INT7/ via OR gate (U34-10) to IR7.
E119 - E114 (9-C3): Connects U15-9 (PFIN/ latch) to INTR5.5.

The following modifications are required for proper PROM type decoding:

(a) Install the single jumper in J7 (4-B6) between pins 6 and 9.

(b) Remove the four-line jumper in J8 (4-B6). Install an 8-pin header in J8 which has three parallel jumpers between pins 9 and 10, 8 and 11, and 7 and 12, respectively. Connect a wire from J8, pin 13, to wire-wrap post E80. Alternatively, a wire connection can be made on the solder side of the board between J8, pin 13, and pin 4 of the 8085 CPU (U36).

These connections simulate a set of four 2716 EPROMs to the address decoder. The address pins A11 of the 2732A EPROMs actually used are connected to the SOD output of the 8085 CPU, which permits ROM bank switching.

Install the iSBC 301 RAM expansion board, and make sure that the maximum RAM address is set to 3FFFH on the 80-24 board (jumper 154-155; default factory configuration). Install the iSBX 331 Arithmetic Processing Unit in Multimodule connector J6 (i.e., in the Multimodule connector in the center of the board). No changes of the default configuration of the iSBX 331 APU board are required. Refer to Intel's iSBX 331 Fixed/ Floating-Point Math Multimodule Board Hardware Reference Manual (order No 142668-001) for further information.

2.3.2 iSBC 064A (or equivalent) Memory Expansion Board

Set the base address of the 64 KByte RAM to 00000H. (This is the factory default configuration of all boards with more than 32 KBytes RAM.) No further connections or modifications are required. Refer to Intel's iSBC 016A/032A/064A/028A/056A RAM Board Hardware Reference Manual (order No 143572-001) for the details of the board setup.

2.3.3 iSBC 517 I/O Expansion Board

Set the I/O base address of the board to 0B0H (jumper pad S2 1-6 and jumpers 87-88). Remove jumper 103-111. Connect the TxRDY and RxRDY outputs of the 8251A USART on the 517 board to the MULTIBUS INT7/ and INT5/ lines, respectively (jumpers 97-105 and 96-107). Make sure the baud rate generator for the USART is wired for 153.6 kHz (corresponding to 9600 Baud),

which is the factory default configuration (jumper pad S1 1-4). Refer to Intel's iSBC 517 I/O Expansion Board Hardware Reference Manual (order No 9800388-01) for further information.

2.3.4 iSBC 204 Disk Controller

Set the I/O base address of the board to 80H (DIP switch S2: 4-off, 5-off, 6-off, 7-on). Connect the Controller's interrupt output to MULTIBUS INT2/ (jumper 63-67). Configure the disk drives as required (compare Intel's iSBC 204 Flexible Diskette Controller Hardware Reference Manual (order No 9800568A), and the documentation supplied by the manufacturer of the drives), and connect them to the controller.

2.3.5 DT772/5716-32DI-B-PGH A/D Converter Board

Configure the board for I/O mapped operation at an 8-bit base address of 20H:

Remove the following jumpers:

W 4	(memory-I/O mode)
W 6	(memory-I/O mode)
W12	(addressing mode)
W33	.
W34	.
W73	(addressing mode)
W36 - W42	(base address)
W 2	(INH1/)
W 3	(INH2/)
W20	(error interrupt)
W21	(data ready interrupt)

Install the following jumpers:

W 5	(memory-I/O mode)
W 7	(memory-I/O mode)
W31	(base address)

All other jumpers should be left at the factory configuration.

Refer to the User Manual for DT772 Series Analog Input Systems for MULTIBUS Computers (document number UM-02829-A) for further information.

2.3.6 MP8316-V D/A Converter Board

The D/A converter board must be operated I/O-mapped, with a base address of 40H. Its output voltage must be unipolar, 0 to 10 volts. The following, and only the following, jumpers must be set on the board:

W 3 *
W 5
W17
W19 *
W21
W23
W25 *
W27 *
W29
W31 *
W33 *
W34
W37 *
W41
W43

(* denotes the default factory setting.)

2.3.7 Cardcage

Assign the highest MULTIBUS priority to the iSBC 204 Disk Controller. Install a mate for the 80-24 board's P2 connector, and connect a momentary action switch (if possible, a switch labeled "INTERRUPT") between ground and P2-19 (PFIN/). This switch causes an RST 5.5 interrupt if pressed, which is used by the ROM resident code to vector control to the Monitor. Advantageous but not indispensable is a "RUN" LED (or a pair of complementary "RUN" and "HALT" LEDs) connected via P2-28 (HALT/).

2.3.8 Console Terminal

Any dumb CRT terminal can be used in the controller computer which allows direct cursor x-y addressing via control sequences with a maximum length of 4 bytes. The actual control codes are determined by a disk file which is loaded together with RXISIS-II (RXISIS.PSC; compare chapter 3.3.4.1.7) and which can easily adapted to terminals with different control codes; the system as implemented at ASU is configured for a

WYSE WY-50 (or a Lear Siegler ADM-5) terminal which uses the following hexadecimal control codes or sequences:

Cursor Up:	[0BH]
Cursor Down:	[0AH]
Cursor Left:	[08H]
Cursor right:	[0CH]
Cursor Home:	[1EH]
Clear Entire Screen:	[1AH]
Clear line:	[1BH] + [54H]

Absolute Cursor Positioning:
[1BH] + [3DH] + [1FH+<line>] + [1FH+<column>]

(<line> and <column> are the intended line and column numbers, starting with 1. The control sequence for positioning the cursor at line 5, column 16 is therefore: [1BH] + [3DH] + [24H] + [2FH].)

Configure the CRT terminal as follows:

Baud rate:	9600
Parity:	odd
Data bits:	7
Protocol:	CTS/RTS

The terminal should permit to send Breaks; it must service the CTS/RTS handshaking lines.

2.3.9 Printer

Any printer with a line length of 132 characters can be used. The printer must be equipped with an RS232 interface and set up as follows:

Baud rate:	9600
Parity:	none
Data bits:	8
Protocol:	CTS/RTS

It must service the CTS/RTS handshaking lines.

Although a printer which can be set to 9600 baud is preferable, any other baud rate can be used if the iSBC 517 board is set up correspondingly (compare chapter 2.3.3).

2.4 Computer - Puller Interface

This chapter is only intended to present a general overview over the interface between the puller and the computer, as far as is required for understanding the operation of the CGCS. Detailed information about the hardware will be given in a separate documentation.

2.4.1 Analog Input Signals

The following analog input signals are submitted to analog-to-digital conversion via the analog input interface circuitry:

* Three Heater Temperatures: Since the puller on which the CGCS was implemented at ASU has only one heater, one hardware interface only has been installed for the heater thermocouple. The measured temperature (or, rather, thermocouple voltage) is routed within the Analog Data Input task of the CGCS to all three heater temperature channels. The input of the interface circuitry is connected directly to the heater thermocouple. The nominal input voltage range is 0 to 30 mV, which corresponds to a temperature range of approximately 20 to 1,500 °C (20 µV ≙ 1 °C).

* Base Temperature: This temperature is measured with a thermocouple located at the center of the bottom of the crucible. The input of the interface circuitry is connected directly to the base thermocouple. The nominal input voltage range is 0 to 30 mV, which corresponds to a temperature range of approximately 20 to 1,500 °C (20 µV ≙ 1 °C).

* Weight: The signal supplied by the weight gauge is preprocessed by the standard analog circuitry of the Cambridge Instruments puller. It is picked up from the input of the analog Automatic Diameter Controller board. Its nominal input voltage range is 0 to 8 V for 0 to 8 kg (1 V ≙ 1 kg).

* Differential Weight: This input signal is obtained from analog differentiation of the above weight signal. This differentiation is done within the analog Cambridge Instruments console by means of a dedicated board (the predecessor of the Automatic Diameter Controller which essentially contains the differentiator circuitry only). Although, on principle, the Differential Weight signal could have been picked up from the Automatic Diameter Controller board as well, this solution was found to be not ideal because any inadvertent or deliberate adjustment on the analog console would have affected the Differential Weight signal measured

by the CGCS. The nominal input voltage range is 0 to 10 V for 1 to 10 g/min (1 V ≙ 1 g/min).

* Seed Lift Speed: This signal is picked up directly at the seed lift tachometer. Its nominal input voltage range is 0 to ±90 V, corresponding to a lift speed of 0 to ±250 mm/hr (1 V ≙ 2.78 mm/hr).

* Crucible Lift Speed: This signal is taken from the crucible lift tachometer. Its nominal input voltage range is 0 to ±90 V, corresponding to a lift speed of 0 to ±20 mm/hr (1 V ≙ 0.22 mm/hr).

* Seed Rotation Speed: This signal is output by the seed rotation tachometer. Its nominal input voltage range is 0 to ±90 V, corresponding to a rotation speed of 0 to ±60 rpm (1 V ≙ 0.67 rpm).

* Crucible Rotation Speed: This signal is generated by the crucible rotation tachometer. Its nominal input voltage range is 0 to ±90 V, corresponding to a rotation speed of 0 to ±66.9 rpm (1 V ≙ 0.74 rpm).

* Seed Position: The Seed Position signal is taken directly from the position potentiometer. Its nominal input voltage range is 0 to 6 V for a seed position range of 0 to 600 mm (1 V ≙ 100 mm).

* Crucible Position: The Crucible Position signal is picked up directly at the position potentiometer. Its nominal input voltage range is 0 to 2 V for a crucible position range of 0 to 200 mm (1 V ≙ 100 mm).

* Gas Pressure: This signal is output by the gas pressure conditioning unit inside the analog Cambridge Instruments console. Its nominal voltage range is 0 to 1.5 V, corresponding to a pressure range of 0 to 1,500 psi (1 mV ≙ 1 psi).

* Three Heater Powers: The Heater Power signals are taken from the power indication output of the heater SCR controllers which are part of the analog Cambridge Instruments circuitry. Since the puller on which the CGCS was implemented at ASU has only one heater, only one hardware interface has been installed for the heater power. The measured value is routed within the Analog Data Controller task of the CGCS to all three heater power channels. The nominal voltage range of the Heater Power signal is 0 to -5 V for a power range of 0 to 60 kVA (1 V ≙ 12 kVA).

* Contact Device: This signal is supplied by the pertinent hardware of the Cambridge Instruments analog console. Its nominal range is 0 to 1 mV, corresponding to a reading on the analog contact device meter of the Cambridge console of 0 to 100 units.

Each of the above 17 channels (only 13 of which are currently implemented at the ASU puller) is measured via an interface schematically outlined in Fig. 5:

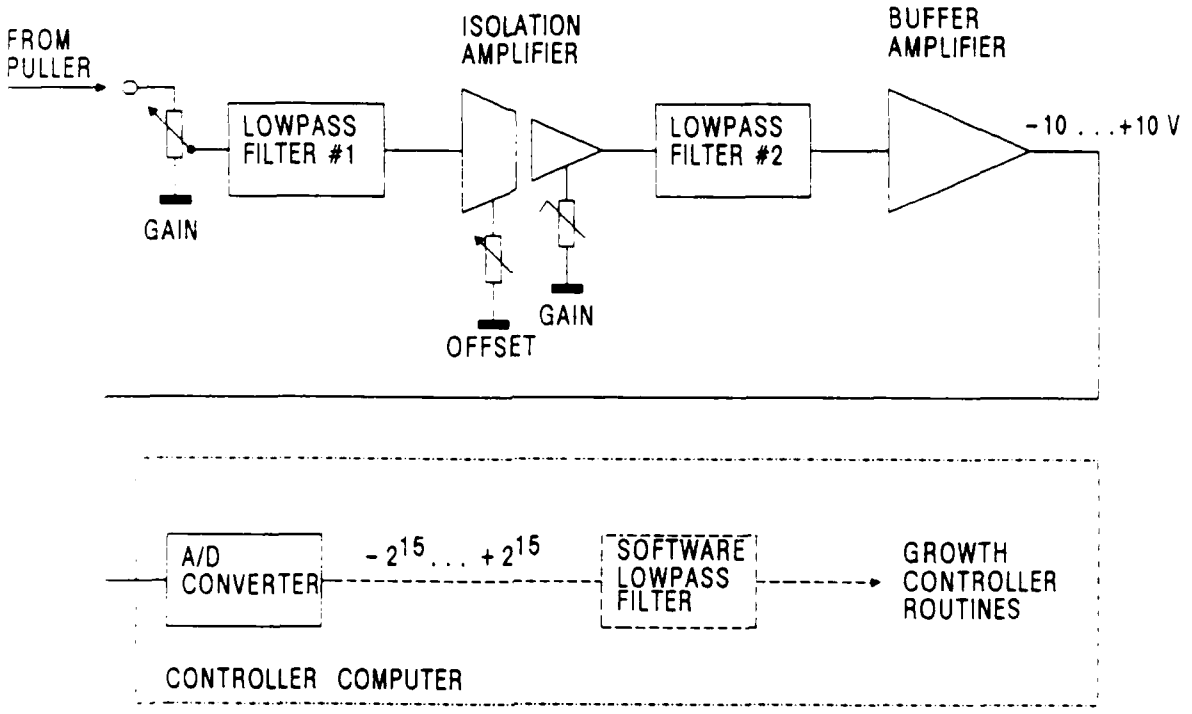


Fig. 5: Analog input interface.

Two generic types of interface boards were developed, one, for the thermocouple signals, and one, for all other sources. (Eight spare channels which were provided for various experimental measurements can be equipped either with a thermocouple, or with a general purpose amplifier.)

The signal generated by the puller is submitted to an isolation amplifier via an input voltage divider which permits a vernier gain adjustment, and reduces the high-voltage input signals from the tachometers to the range of ±10 V permitted by the isolation amplifiers. A low-pass filter blocks possible r.f. noise which might cause interferences in the isola-

tion amplifiers. (No vernier gain adjustment was provided for the thermocouple amplifiers.) The isolation amplifier prevents ground loops and the interferences which are usually caused by them, and it provides an easy way of inverting the polarity of signals like the heater power voltage. Its gain is set such that the voltage at the output of the isolation amplifier is in the range of -10 V to +10 V. This output signal is submitted to a second low-pass filter with a cutoff frequency in the order of 0.1 Hz; this filter is required for eliminating components with a frequency greater than half the sampling frequency (1 Hz) which might cause aliasing otherwise.

The conditioned signals from all analog input channels are submitted to a 32 channel, 16 bit A/D converter which is physically part of the controller computer. The A/D converter is configured to transform analog signals in the range -10 V ... +10 V to signed integers between -32,768 and +32,767. The A/D converter operates under the control of the CGCS software; it is programmed to read each active input channel once per second in a random access mode. The relation between the physical input channels of the A/D converter and the logical data used by the CGCS software is determined by a software-based table which links each element in an input data array to one hardware channel of the A/D converter. This table, and even its size, can be modified easily, even while the CGCS is running, which permits an extremely flexible operation if hardware channels have to be activated or de-activated temporarily for some experiments.

Before the data read from the A/D converter are made available to other tasks within the CGCS, they are submitted to digital low-pass filtering. In contrast to the hardware filter circuitry, the software filter routines can be re-programmed easily; this filtering further reduces random noise and spikes. This is particularly important if not each measured input value is actually used by the CGCS algorithms, which applies, for example, to all input data required for the diameter evaluation. The low-pass filters provide in this case a weighted average over the recently measured data, rather than the plain (and possibly noisy) measured values.

2.4.2 Analog Output Signals

The following signals have to be supplied to the puller by the digital controller:

* Three Heater Power Setpoints: These output signals replace the output of the analog temperature controllers within the analog console; they are connected directly to the inputs of the heater SCR controllers. Only one Heater Power Setpoint channel is currently used in the single-heater system at ASU. The voltage range required is 0 to 5 V, corresponding to 0 to 100 percent heater power.

* Four Motor Speed Setpoints: These signals replace the set-point voltages of the analog controller for the seed and crucible lift and rotation speeds. Their required voltage range is 0 to 10 V for the rotation speeds, and 0 to 5 V for the lift speeds.

The analog output circuitry is schematically shown in Fig. 6.

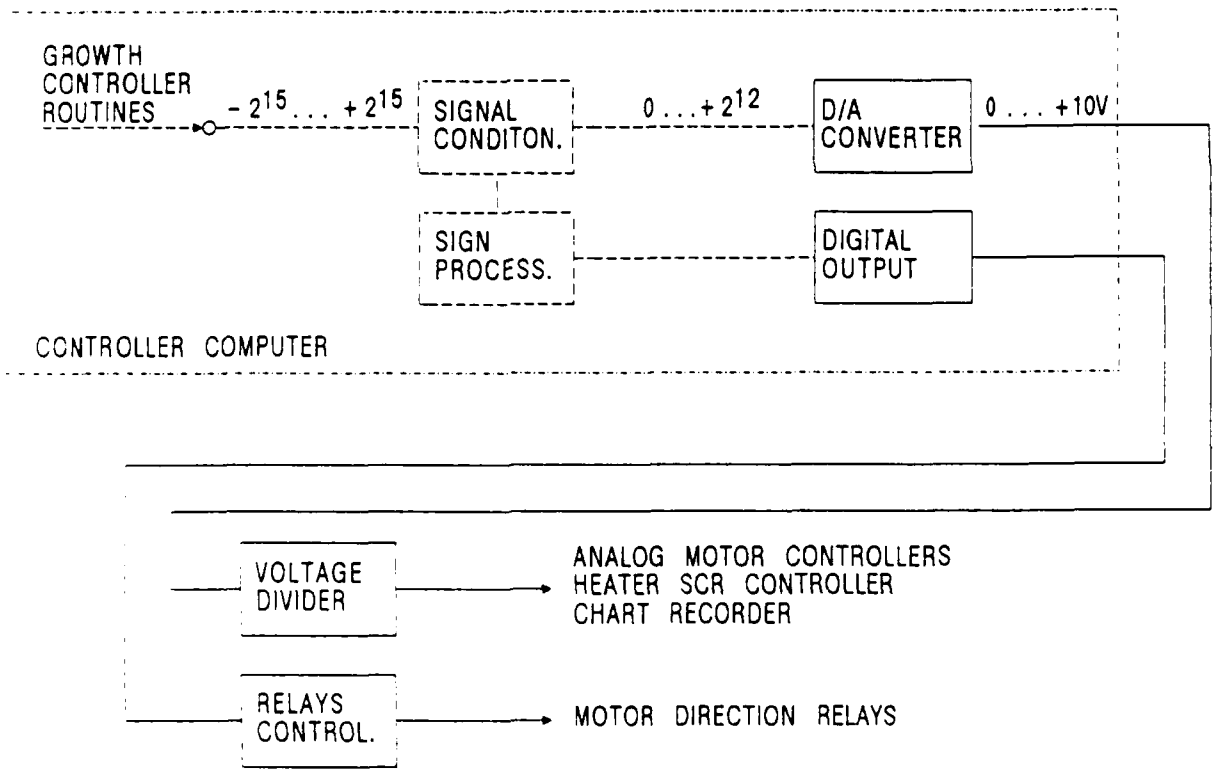


Fig. 6: Analog output interface.

The 12-bit D/A converter used was set to unipolar output since only positive voltages are required by the puller. Digital data supplied to it by the Analog Data Controller task have therefore to be converted to their absolute values, and scaled to a 12-bit range (0 to 4095). While the heater powers are intrinsically positive data, this does not apply to the motor speed setpoints whose signs determine the direction of the movement (up/down or clockwise/counterclockwise). The signal conditioning routine strips therefore the signs off the motor speed setpoints, and supplies them to a digital output routine which activates the relays which control the motor directions accordingly. (It should be noted that the measured motor speed data are bipolar, while the pertinent setpoints must be unipolar.)

The output signals generated by the D/A converter are connected to the corresponding inputs of the puller either directly, or via simple voltage dividers. It was not found necessary to install isolation amplifiers for the output signals, firstly, because their voltage levels are relatively high, and secondly, because they are in a less sensitive part of the control loops.

Eight channels of the D/A converter are hooked up to a chart recorder; no special interface is required for them either. The signals supplied to the chart recorder are set to their absolute values by the CGCS software; a message is issued at the CGCS console if a parameter which is being routed to the analog output changed its sign.

2.4.3 Digital Input and Output

The sole application of digital input signals is monitoring of the status of the motor direction control relays inside the Cambridge Instruments analog console. These relays are controlled by switches on the analog console if the analog controller is in charge, or, otherwise, by digital output generated by the CGCS. Since one of the constraints of the CGCS was that a "bumpless" transition between the analog and the digital controllers should be possible under all circumstances, it was necessary to monitor this relays status continuously. This information is used to preset the motor speed setpoints when the CGCS gains control; the digital controller provides output signals for the motor control relays which replace the ones derived from the front panel switches on the analog console.

Status input and control output is mapped to two lines for each motor, which corresponds to one input and one output byte, respectively. One additional digital output line controls a changeover relay which allows to use control input from the analog console if it is de-activated, and from the digital controller if it is energized.

The digital outputs are routed to relays on a digital interface board which generate the actual control signals for the Cambridge Instruments circuitry, and which provide full isolation between the 5 V logical level within the controller computer, and the 28 V rectified a.c. voltage used by the Cambridge console for relays control. Similarly, the relays status of the Cambridge console is converted into signals suitable for digital processing by a number of relays on the digital interface board.

3. System Software on the CGCS Computer

3.1 Design Considerations for a Real-Time Operating System

3.1.1 Intel's iRMX-80 and FORTRAN

From the software point of view, there are various approaches for designing real-time process controllers: The most simplistic one is writing a dedicated program which has to comprise all auxiliary functions like console and data I/O, and which is able to react properly to asynchronous external events (like an operator command, or a machine status change) by the use of hardware interrupts, or by means of software polling loops. For a more complex environment or process to be controlled, however, this approach is hardly feasible any more. The programmer is not only overburdened with all the auxiliary and housekeeping functions whose number and complexity increase dramatically as the complexity of the system operations grows; the controller software proper also tends to get out of hand and become confused due to the lack of modularity which is frequently found with such systems.

A considerable improvement of the readability and serviceability of a process controller or any other real-time computer application can be achieved by using a special real-time operating system. An operating system off-loads the programmer from chores like providing software drivers for peripheral devices; it usually hides the genuine constraints of real-time operations from the programmer, permitting him to concentrate on a functional rather than chronological approach for the layout of the controller software. Breaking down the operation of a system into a number of more or less autonomous "tasks" is a first step towards modular programming; since tasks can be coded rather independently (even by different programmers), program development is facilitated, and corrections of errors or software modifications become easier.

For the 8-bit, 8080/85-based, hardware chosen, a real-time operating system has been distributed by Intel Corporation under the name "iRMX-80" (Intel's Real-Time Multitasking Executive for 8080 or 8085 processors). Although already obsolete when finally applied for the design of the CGCS, iRMX-80 provides absolutely sufficient support for an application like the Czochralski controller, at least, with a number of specially written enhancements.

The development of software for an 8-bit environment was always somewhat impeded and limited by the restricted availability of programming languages. Intel supported - at least at the time when the CGCS and its supporting software were de-

signed - only three high-level languages in addition to assembly language, namely, BASIC, PLM-80, and FORTRAN.

As an interpreter-based language, BASIC had to be ruled out immediately for an implementation of a more complex system like the CGCS since it could not be combined reasonably with the functions of iRMX-80 to constitute a genuine real-time system. There is an iRMX-80 based version of BASIC which does, however, not allow to break down a complex system into a number of BASIC coded tasks; its interface to modules programmed in different languages is more than awkward. The size of the BASIC interpreter and, even more, its insufficient speed virtually prohibit its use for any serious application. BASIC is a valuable tool, though, for the setup and testing of various I/O interfaces, and for the fast development of small auxiliary programs; two BASIC versions were therefore prepared to be run on the CGCS computer.

PLM-80 (a dialect of PL-I for the microprocessor environment) was used and recommended by Intel as an implementation language for iRMX-80; accordingly, the interfaces to the iRMX-80 system routines were designed for a call by PLM-80 coded modules. PLM-80 has a major drawback, though: Its numeric routines support only integer variables in the range of -32,768 to 32,767; the floating-point operations which were, in fact, found to be required for most of the CGCS modules would have to be coded with awkward calls to routines of a floating-point library.

FORTRAN-80 (Intel's implementation of FORTRAN 77), finally, was the only language supplied for the 8-bit environment which provides floating-point operations as a standard. FORTRAN was therefore the obvious choice for at least those routines of the CGCS which use floating-point variables. Although the program development environment provided by Intel would have permitted to freely combine modules coded in FORTRAN-80, in PLM-80, and in assembly language, it was not considered wise to use both high-level languages: Either of them requires a separate set of supporting library routines (although FORTRAN-80 shares some of the PLM-80 libraries), which would have increased the total size of the program code unduly. Since the capabilities of PLM-80 are essentially a subset of those supported by FORTRAN, FORTRAN-80 was chosen as the basic implementation language of the CGCS.

Unfortunately, FORTRAN-80 uses a parameter passing convention which is, in general, not compatible with PLM-80 and, hence, iRMX-80; interface routines had therefore to be provided which permit to call iRMX-80 system routines from FORTRAN. These routines were generally written in assembly language with

regard to speed and code size. Furthermore, the standard FORTRAN-80 I/O routines turned out to be practically unusable for the application in mind: FORTRAN I/O is extremely slow and requires an excessive amount of program code and stack resources. In addition, it does not reasonably support the generation of a fixed console screen mask which was found indispensable for the continuous display of a large number of measured values. These considerations led to the development of dedicated routines for console, printer, and disk I/O which require only a fraction of the system resources needed by the corresponding FORTRAN-80 routines. These I/O routines were coded in assembly language as well in order to guarantee a sufficient performance. The obvious penalty of using special I/O routines was, however, that the resulting code can hardly be regarded as standard FORTRAN.

A further enhancement was provided by replacing the standard software-based floating-point libraries of FORTRAN-80 by specially written interface routines which use the numeric processor hardware, i.e., the 8231 Arithmetic Processing Unit chip on the iSBX-331 expansion board. With a small deterioration of the achievable accuracy, these routines increase the execution speed of floating-point operations by about one order of magnitude, and simultaneously decrease the size of the required floating-point program code to about 50 percent.

Within this chapter, the general rules for coding FORTRAN tasks in an iRMX-80 environment shall be reviewed. The discussion comprises the following points of view:

* The structure of a task in a FORTRAN-iRMX-80 environment.

* Sharing of common code sequences between several tasks.

* Data transfer between tasks.

* Generation of RMX control structures in a FORTRAN based system.

* Input and output of data in a real-time environment.

For basic information about iRMX-80 and FORTRAN-80, the reader should refer to the pertinent documentation supplied by Intel Corporation.

3.1.2 The Structure of a Task in a FORTRAN-iRMX-80 Environment

No user-supplied main programs are permitted in an iRMX-80 environment. The attempt to execute a main program under iRMX-80 will result in a disastrous system error. The main routine of each task must therefore be coded as a subroutine in FORTRAN; its general structure comprises an initialization sequence which is executed only once when the task starts running the first time, and an endless loop. There must not be any exit from this loop, particularly not via a "RETURN" statement, except if a task suspends or deletes itself. The subroutine forming the main body of the task may in turn call other subroutine or function subprograms; special care must be taken, though, if a routine may be used by more than one task.

The task initialization sequence depends on which operations are to be performed by the task. It must contain a call to the routine FQFSET if the task uses any floating-point operations or intrinsic functions, a call to the routine FXIOST (compare chapter 4.2.2.1) if the routine performs any I/O operations via the special routines rather than via FORTRAN "READ" or "WRITE" commands, and a call to FRINIT (compare chapter 4.2.1.2) for each message - response exchange combination which is permanently allocated to the task. Furthermore, all other exchanges including interrupt and "flag interrupt" (a special enhancement added to iRMX-80, compare chapter 4.2.1.4) exchanges which should be built by the task must be created during this initialization sequence. A FORTRAN task which uses the first three functions may therefore have the following structure:

```
        SUBROUTINE MYTASK
        INTEGER*1 MYEXCH(19), IOEXCH(31)
        COMMON /MYTSK1/ MYEXCH,X,Y,Z,A,B
C           (The named COMMON block comprises 19 bytes
C           reserved for an exchange-message combination,
C           and 5 REAL variables with 4 bytes each.)
        CALL FQFSET (0,0)
C           (Compare FORTRAN-80 Compiler Operator's Manual)
        CALL FXIOST (IOEXCH)
C           (This subroutine call initializes the I/O
C           structures; compare chapter 4.2.2.1)
        CALL FRINIT (MYEXCH,20,199)
C           (This statement has the effect that the next 20
C           bytes following MYEXCH are incorporated in
C           a message with TYPE 199; compare chapter 4.2.1.2)
100     .....
C           (The main task code follows here)
        GOTO 100
```

3.1.3 Sharing of Common Code Sequences Between Several Tasks

Coding in a real-time environment differs significantly from the straightforward approach which is possible with batch execution. In general, there is no possibility to predict which task will run at which time or will be going to access data or shared software resources. The basic philosophy of iRMX-80 is that separate resources are allocated to each task, in particular, a separate stack. Whenever a task is interrupted or waiting at an exchange its current status (i.e., the status of the processor's hardware registers) is saved on the task's stack. Therefore, data is automatically protected if it is kept on the stack or within memory locations which are local to the task (which means that they cannot be accessed by any other task). A variable which is not kept in a COMMON block is intrinsically local to a FORTRAN routine.

This protection does not hold, however, if a subprogram is shared by several tasks and if this routine uses locations in read-write memory for intermediate data storage. If a task has been interrupted while executing such a common routine it will resume execution exactly at the point where it was interrupted when it becomes ready again; still, the local data within the common routine may have been changed meanwhile by another task which used the same routine. There are several possibilities to avoid such unpredictable occurrences: The common routine may be linked separately to each task which uses it, it may be compiled with the "REENTRANT" compiler option, it may disable interrupts during critical operations, or a software lock may be applied to it.

The first approach may appear to be the easiest one, still, it expands the memory requirements for the code significantly, particularly if the routine is rather lengthy (and even relatively simple FORTRAN routines turn out to require plenty of code when they are compiled).

The second approach permits the mutual use of a routine by several tasks as data is allocated on the task's stack rather than in absolute memory locations if reentrancy was specified. Still, this method extends the stack requirements of each task which calls the common routine; stack requirements are already rather high anyhow in a FORTRAN environment. In general, the FORTRAN library routines, e.g., for floating point arithmetics and intrinsic functions, are reentrant, and so are most of the iRMX-80 modules. Storing local data on the tasks' stacks, they add significantly to the stack requirements. The standard FORTRAN-80 I/O routines, for example, require an additional stack of 800 bytes for each task which performs I/O. Besides allocating data on the tasks' stacks, the FORTRAN-80

floating-point routines use locations in read-write memory which are intrinsically private to each task because they are located within an extension of its task descriptor. (A similar approach is also applied by the specially written I/O routines; compare chapter 4.2.2.)

Routine protection by disabling all interrupts of the system constitutes the least code and execution time consuming protection approach. A routine which is protected in this way can never be interrupted since external events (including system clock ticks) simply cannot be recognized by the operating system nucleus (which is in charge of the interrupt handling); no other task can become active therefore while interrupts are disabled, no matter what its priority is. However, this protection technique cannot be used for lengthier code sequences because it would unduly deteriorate the reaction of the system to external events, and it cannot be used on principle if the protected code tries to invoke operating system functions. Its application is therefore limited to sufficiently fast processes which do not require iRMX-80 routines; the high-speed floating-point routines for the numeric processor (compare chapter 4.2.5) were designed to use protection by interrupt disabling.

The last method of protecting a common routine, namely, by means of software interlocks, imposes a slightly increased execution time overhead; it may also affect the order of execution of the tasks. The software interlock makes any task wait at an exchange immediately at the beginning of the common routine if the routine is being used by another task. No matter what its priority is, the task has to wait until the execution of the protected sequence by the currently running task is terminated. When leaving the common routine, the currently executing task sends a message to the entry exchange which permits the first task which waits there to become ready and to resume execution if no task with a higher priority is ready. Such a software interlock can be easily accomplished by calling the subroutine FRACCS (compare chapter 4.2.1.5) prior to entering the protected sequence (i.e., prior to the subroutine or function call which accesses the common code), with an exchange exclusively used for the protection of the particular common code sequence as a parameter. The next statement after the call to the shared routine must be a call for the subroutine FRRELS, specifying the same control exchange, which releases the software interlock.

In order to compare these approaches, consider the following situation: a low-priority task is just executing a routine shared by other tasks when a high-priority interrupt handling task becomes ready which will eventually access the same

common routine. In the case of separate copies of this routine for both tasks, the interrupt routine will run regularly, without any additional delay or overhead. The same applies if reentrancy was specified for the shared code; the parameters of the low-priority task are saved on its stack, and the resources of the common routine can be fully utilized by the high-priority task. The high-priority task will never become ready if interrupts were disabled by the common routine, but a second interrupt may be missed if it happens before the protected routine re-enables the interrupt system. In the case of a software interlock, the interrupt handling task finds itself waiting at an exchange where there is no message available; it is therefore removed from the ready list, and the task with the highest priority which is ready becomes the running task. This task may or may not be the task which is blocking the common routine. Anyhow, the interrupt routine has to wait until the low-priority task has terminated its execution within the shared code sequence and has sent a message to the exchange guarding the entry point of this code. Only then, the interrupt task will be returned to the ready list and probably become the running task. The unpredictable delay imposed upon the interrupt task may, however, have caused the missing of an interrupt. The use of software interlocks should therefore be considered very carefully, and they should generally not be used in conjunction with interrupt handling tasks or with tasks with a very critical timing. (As a matter of fact, this does not only apply to the explicit use of software interlocks. Some routines, for example the special I/O routines of chapter 4.4.2, may also impose an undue delay on a task with critical timing.)

3.1.4 Data Transfer Between Tasks

Similar considerations apply to the exchange of data between tasks. There are two main ways for passing data: first, memory locations may be used for data storage which can be accessed by several tasks. In FORTRAN which does not support the "PUBLIC" scope of labels and variables, the only way to do so is using a (named) "COMMON" block. The second possibility for performing transfer of data is to send it to another task, formatted as a message.

Both approaches have advantages and disadvantages: Data transfer via a "COMMON" block must be protected by a software interlock; some of the approaches - interrupt disabling or a software interlock - which can be used for protecting common code sequences can be applied for this purpose as well. This protection is indispensable because you never can predict the

occurrence of an interrupt. It might happen just when a task is reading a multi-byte variable in a "COMMON" block, and it might trigger the execution of another task which accesses the same data area. The interrupting task may or may not change the variable which was just being read by the interrupted task. When the interrupted task resumes its execution, the second part of the variable it is reading may significantly differ from its previous value although the variable itself was only slightly changed. (For example, a two-byte integer may have held 256 (0100H); its value is to be changed by an interrupt triggered task to, say, 255 (00FFH). The interrupt might happen between the reading of the low byte (which is still at 00H) and the high byte (which is set from 01H to 00H by the interrupting routine). The interrupted routine will therefore read a value of zero (0000H) instead of the correct values 256 or 255 which it would have read had the interrupt occurred a few microseconds earlier or later.) Still, the same considerations concerning the detaining of high-priority routines apply as for the protection of shared code by a software interlock.

A software interlock is not necessary for passing data through COMMON blocks or other shared data structures in read-write memory, though, if a proper operation can be guaranteed by prudent choice of the priorities of the routines involved: If data is to be written by the task with the lowest priority only (at least, with the lowest priority among the tasks which access the data), this task can never interrupt any other task because its priority is too low. Updating of a data location can therefore never interrupt any read access to the same location. The low-priority task writing the data ought to disable interrupts during its writing operation, though, to protect itself from being interrupted during this critical operation. Since writing two or four bytes requires only a matter of microseconds, the delay imposed on the interrupt response is negligible which is introduced by disabling the interrupts. This approach is used within the CGCS for updating arbitrary memory locations addressed by symbolic names ("Variables"); no other protection method was feasible for this purpose without unduly restricting the range of accessible memory locations.

Although sending a message implies a significantly higher software overhead, it appears to be the safer way, particularly for interrupt service routines, and it is therefore the only genuine iRMX-80 data passing technique. The major disadvantage of the message approach is, however, that data can be sent to only one other task; in contrast, the use of "COMMON" blocks permits access to the data by an unlimited number of tasks.

There is an important exception to the need of data access protection in a real-time environment: Single bytes can be freely written or read without any further software overhead. This is true because the current operation is always terminated by the hardware before an interrupt is acknowledged and serviced. Single bytes will therefore always hold a correct (but possibly slightly obsolete) value. This applies in particular to Boolean variables (flags) which can therefore even be used for a software "interrupt" (compare chapter 4.2.1.4). (Incidentally, word type variables (e.g., two-byte integers) may also be transferred without protection if they are always referred to by their absolute addresses, and if no address calculation is involved. These variables are, in general, also moved with one machine code instruction only. These considerations apply in FORTRAN to simple INTEGER*2 variables, and to elements of INTEGER*2 arrays which are referred to by explicitly coded subscripts. J(3) and N do not need protection therefore if J and N are of type INTEGER*2; however, J(N) does since array elements whose subscripts have to be calculated at execution time are transferred by FORTRAN-80 in a byte-by-byte mode.)

3.1.5 Generation of Control Structures in a FORTRAN-based iRMX-80 System

The peculiar properties of FORTRAN, particularly the limited scope of variable names, require some special approaches for the generation of and the access to iRMX-80 control structures by FORTRAN routines. Some of these structures have to be supplied by dedicated PL/M or assembly language modules in any case.

3.1.5.1 Static Task Descriptors and Task Descriptors

The most straightforward approach for generating these structures is including them into the configuration module. This module must either be coded in PL/M or in assembly language, or it can be interactively created by means of Intel's Interactive Configurator Utility (ICU-80). Still, it should be noted that the ICU-80 software is only capable of creating Static Task Descriptors kept in the configuration module; it is indispensable to use dedicated assembly language or PL/M code for creating Static Task Descriptors if they are, e.g., to be loaded from disk at execution time. (Special interface routines have been written, though, which permit to use an

ICU-80 generated Configuration Module within an iRMX-80 system overlay; compare chapter 3.4.5).

Additional care is required when the Static Task Descriptors of a FORTRAN-iRMX-80 system are defined: FORTRAN requires an extension of the Task Descriptor where it can place its floating-point registers. (This approach provides individual floating-point registers for each task. Although the FORTRAN floating-point routines require such locations in conventional data memory they can therefore be considered reentrant.) The length of this extension depends on the type of floating-point arithmetic used: 18 extra bytes have to be reserved if software floating-point routines or the special routines for the 8231 Arithmetic Processor Unit are used, 13 bytes, if the system contains an iSBC 310 High Speed Mathematics unit and the corresponding FORTRAN libraries are included.

Two more bytes in the Task Descriptor extension area are needed if the task performs I/O via the special routines mentioned above. These extra bytes have also to be declared in the "EXTRA" line of the Static Task Descriptor building program sequence if the Interactive Configurator Utility is used to create these control structures.

3.1.5.2 Exchange Descriptors

Exchanges may be generated at any time by a FORTRAN task with a call to the interface routine FRCXCH (compare chapter 4.2.1.3). A sufficient area in read-write memory is required where FRCXCH can build an exchange, i.e., ten contiguous bytes. The way how to supply this memory area depends on the intended scope of the exchange: an exchange which is to be used within the task only, e.g., for a timed wait in a timer task, or for task suspending, may be specified by the following statements:

```
        INTEGER*1 EXCH(10)
        CALL FRCXCH (EXCH)
100     ...
C           (Infinite loop of the routine)
        GOTO 100
```

In this case, the scope of the exchange EXCH is limited to the subroutine where it was defined. If this routine calls other subroutines, it has to pass EXCH as a parameter in order to permit access to the exchange built at this memory location. Still, some types of exchanges do not require a global scope altogether although they are accessed by other tasks. This

applies in particular to response exchanges, used in conjunction with FRSEND (compare chapter 4.2.1.2), and to "flag interrupt" exchanges generated with a FXCRFE call (compare chapter 4.2.1.4). The information about these exchanges is transferred by a message to the tasks which are supposed to send messages to them.

The probably more common use of an exchange is its application for data or control transfer between different tasks. Such an exchange can no more be built in normal data storage locations as there is no way to let other tasks know about its position. The only possibility for creating such exchanges is to build them in memory allocated to a (named or unnamed) "COMMON" block. Exchange-message combinations which control the access to data within a "COMMON" block may (but need not) be created within this "COMMON" block, for example in its first locations. Other exchanges may be contained within one or more specialized "COMMON" blocks (which need not be protected by a software interlock as they are only accessed by iRMX-80 routines). The declaration sequence of a task routine using such exchanges might be:

```
INTEGER*1 EXCH1(10),EXCH2(10),EXCH3(10)
COMMON /EXCH/ EXCH1, EXCH2, EXCH3
```

Note: An exchange must be created once and only once during program execution. Creating an exchange twice may cause a disastrous system error if tasks are already waiting at the exchange. Particularly exchanges located in "COMMON" blocks must be treated very carefully: they must have been initialized when the first message is sent to them or when a task wants to wait there for a message, but only one task can be responsible for creating them. The safest way to build exchanges properly is either to specify them in the configuration module, or to let them be created by a dedicated initialization task or subroutine which should have a sufficiently high priority to run before any other task which might use the exchanges, or before other tasks have been created altogether; the latter approach is used in the CGCS.

In order to avoid a lengthy and complex configuration module or common initialization routine, only these exchanges should be created beforehand which might be accessed by more than one task. An exchange which is used by one task only might as well be created by this task, during its initialization sequence. This approach can help to improve the clearness of the software structure.

3.1.5.3 Messages

The structure of a message within an iRMX-80 system depends essentially on its particular purpose: Some messages are only required in order to trigger the execution of a task (e.g., the interrupt messages), some of them are used to transfer data. Some messages should be sent back to the transmitting task to acknowledge their receipt or to indicate the termination of some kind of processing. This multitude of different task structures was probably the reason why there is no message-creating iRMX-80 routine. Creating a message in FORTRAN, however, may come close to impossible if the message should not only contain data but also addresses. The process of building messages was therefore incorporated into special message sending and receiving routines (compare chapters 4.2.1.1 and 4.2.1.2); the only prerequisite of these routines is that data which are to be transmitted with the message must be kept in contiguous memory locations.

This can be accomplished in two ways: either can the data be located in a COMMON block, in which case the sequence of variable names in the COMMON statement defines the sequence of storage locations in memory; or variables may be defined locally and forced into a certain order by means of an EQUIVALENCE statement. The use of a COMMON block for this particular purpose is not recommended: The COMMON block should not be accessed by any other task if it contains message data (although a software interlock might be used for its protection under certain circumstances), and declaring a named COMMON block means that its name has to be kept reserved for the entire system, which might make programming a little more difficult, aside from the fact that each COMMON block needs a special treatment during linkage and locating. Anyhow, the following examples show how data even of different types can be arbitrarily arranged in memory. Suppose four variables, two REAL (4 bytes long), one INTEGER*1, and one INTEGER*2, should be stored contiguously. This can be done with a COMMON block:

```
REAL A,B
INTEGER*1 I
INTEGER*2 J
COMMON /MESSG1/ I, A, B, J
```

The above sequence of statements has the effect that the first byte of the common block holds I, the next eight contain A and B, respectively, and the last two of the eleven bytes, J. These eleven bytes can be referred to by specifying I as a parameter in the corresponding routine calls. The second

approach requires a little more coding but confines its effect to the routine where the block is declared:

```
      REAL A,B
      INTEGER*1 I
      INTEGER*2 J
      INTEGER*1 DUM(11)
      EQUIVALENCE (DUM(1),I),(DUM(2),A),(DUM(6),B),
*                 (DUM(10),J)
```

The four variables are forced into contiguous memory locations by setting them equivalent to elements of the dummy array DUM whose elements are, of course, intrinsically contiguous. The resulting data block can again be referred to as I or - if preferred - as DUM or DUM(1). (FORTRAN-80 defaults to the first element of an array if the subscript is omitted.) Great care must be taken, however, to assign the correct locations within the dummy array to the corresponding variables as there is no checking whatsoever which could detect a possible overlap or gap. The memory consumption and the execution time of the COMMON and EQUIVALENCE approaches are identical.

The special FORTRAN-iRMX-80 interface comprises two different sets of routines which permit to perform the "send", "wait", and "accept" operations with a call from a FORTRAN program. One set - FXSEND, FXWAIT, and FXACPT (compare chapter 4.2.1.1) - copies the contents of a specified data block into memory supplied by the Free Space Manager, sets the additional parameters, sends the message, copies (within the receiving task) its data contents to a similar memory block which belongs to the receiving task, and returns the memory holding the message to the Free Space Manager. This set permits the unlimited queuing of messages at the receiving exchange; it implies therefore a first-in-first-out buffer operation. Its routines are interlock protected and therefore not recommendable for high speed tasks (aside from the fact that the Free Space Manager might impose an indefinite delay if it runs short of memory). For applications with a critical timing, a second set of routines has been provided, namely, FRCRSP, FRSEND, FRWAIT, and FRACPT (compare chapter 4.2.1.2). Within this set of routines, only one message is used for a given data interface. This message is allocated within the local memory of the sending task, together with a response exchange. Before any data is placed into the message, the sending task must make sure that the message has already been returned by the receiving task, calling the LOGICAL*1 FUNCTION FRCRSP. If the message is available, it can be modified and sent to the receiving task. The receiving task copies the message's contents to its own data area and sends the message back to the transmitting task. As all these routines are reentrant and,

as the data transmission is simply skipped if the receiving task did not yet acknowledge the receipt of the previous message, the tasks involved cannot be delayed. The penalty for this is the possible loss of information, and the lack of a buffering feature. Still, the loss of information will usually not matter if it affects only data which are updated periodically.

3.1.6 Data I/O in a Real-Time System

The program-user interface of a real-time process control system differs significantly from the one applied to a batch processing program: An ordinary non-real-time interactive program may request input, wait until the operator (or the disk controller) has supplied this input, continue execution, and write its results to an output device. This sequence may be repeated infinitely, still, the execution of the program will always follow the same scheme. In a real-time environment, however, processing is (usually) not totally suspended while a task waits for input. A number of other tasks may be executed concurrently, and some of them may generate output. The straightforward data I/O approach used in batch processing - request for input and output of a result in a fixed consecutive mode - does not hold any more in a real-time system. If several tasks are executed in parallel (or, as a matter of fact, consecutively but with undetermined order), they create also output in parallel, and they require input in parallel rather than serially. The I/O routines supplied with FORTRAN, however, support only a serial input and output. Therefore, special I/O routines were developed which permit random disk output, and quasi-parallel console output. The latter is accomplished by random access addressing of the console CRT screen, which permits to write a particular output item (e.g., a measured parameter) always to the same location on the screen. The output area on the console screen can be subdivided into a scrolled and a non-scrolled part to allow random access output (in the non-scrolled portion), and the display of basically sequential data such as the echoes of the input entered on the console (which must necessarily be sequential) in the scrolled area.

While the generation of output by a number of different tasks does not create any problems (it might even be helpful if the amount of output generated varies strongly with time since using several output tasks would permit some data buffering), input cannot be performed unambiguously by more than one task. Generally, the operator has to be notified which data is expected from him by the system, which is done with an output

action. Even if the task that has written the input request line to the console waits immediately afterwards for the data input there is no guarantee whatsoever that there will not be an interrupt in between which, in turn, might make another task issue an input request, possibly even before the first request could be noticed by the operator. Therefore, only one task within the system may perform all the data input and the output of input requests. (In the CGCS, this is done by the Command Interpreter task.)

A similar philosophy applies to the disk I/O. Although the special FORTRAN-iRMX-80 interface routines can handle random disk files and although iRMX-80 allows for one file to be opened for reading by more than one task, there are some important practical restrictions. In general, only one task should be responsible for the input from a disk file to make sure that the contents of the file are processed correctly, and only one task is permitted to write to the disk. Since disk accesses can require relatively long times to be processed it might be a good idea to perform data collection and data output within separate tasks. This approach was, in fact, used in earlier versions of the CGCS for the output of measured data to a disk file; the undue consumption of Free Space Manager-supplied memory by this approach demanded finally to abandon it from version 2.2 on.

3.1.7 Naming Conventions

In order to avoid a collision of the variable and routine names between the iRMX-80, FORTRAN-80, and interface routines which are kept in various libraries, and between system and actual application routines, special conventions for the names of program modules and PUBLIC entry points and data locations have been defined:

* iRMX-80 routines use names beginning either with "RQ...." or with "R?....". Some of the alternative iRMX-80 routines which were provided for an enhancement of the operating system (compare chapters 3.3.4 through 3.3.6) use PUBLIC variable names beginning with "R@....".

* FORTRAN library routine names begin either with "FQ...." or with "F?....". Entry points internal to the special floating-point routines based upon the 8231 APU (compare chapter 4.2.5) have names starting with "F@....".

* Variable and constant names declared PUBLIC by the FORTRAN-iRMX-80 interface routines begin with "F0....".

* The names of reentrant FORTRAN-iRMX-80 interface routines start with "FR....".

* FORTRAN-iRMX-80 interface tasks, non-reentrant routines, and exchanges have names beginning with "FX....".

3.2 Software Structure

As already mentioned above, the design of the controller computer, and the choice of its operating system environment, was directed towards a maximum of flexibility and ability of the system to be run in a stand-alone mode. This entails that the major part of the software which is to be executed on the controller computer should be loadable from a mass storage device only if and when required. Accordingly, only those modules were designed to be kept in Read Only Memory (ROM) which are absolutely indispensable for the operation of the system, i.e., the Nucleus of the operating system iRMX-80, a Loader task which runs under iRMX-80 and which allows to load programs from disk, and the Terminal Handler which constitutes the interface between the software executed on the controller computer and the console terminal. (The latter function was, in fact, not required for boot-loading programs, but it appeared reasonable to keep a routine in ROM which was presumably required by each application.)

The possibility to split a total of 16 KBytes ROM into two banks of 8 KBytes each could advantageously be used for the implementation of auxiliary programs like a Monitor which constitutes a simple but powerful debugging tool, and a set of hardware Confidence Test routines. These functions do not necessarily require the support of iRMX-80; hence, they were put into one 8 KBytes bank of ROM (Bank 0), while the iRMX-80 routines reside in the second bank, Bank 1. Since only one of the two banks may be active at a given time, the use of the Monitor or Confidence Test routines precludes access to iRMX-80, and vice versa.

After power-on or a reset, the system first activates ROM Bank 0 (via the SID output of the 8085 processor), and executes a memory test routine which is part of the Confidence Test. If the memory test was passed without an error, the operator is given the choice of either entering the Monitor routines, or of loading a disk resident operating system, RXISIS-II.

RXISIS-II is, in fact, an iRMX-80 based task which emulates the operating system ISIS-II (Intel's System Implementation Supervisor) (hence "RXISIS"). ISIS-II is the operating system developed by Intel Corporation for its Series-II Microprocessor Development Systems; it comes with a number of utility programs some of which are indispensable for the stand-alone operation of the CGCS computer, and it is a favorable environment for the development of additional utilities and auxiliary programs. ISIS-II provides functions like directory-based access to disk files, and unified input and output to devices and disk files, to programs running under its supervision;

RXISIS-II was accordingly designed to duplicate all essential support functions within the iRMX-80 environment, essentially by re-formatting the parameter tables of the ISIS-II system calls and forwarding them to iRMX-80.

When RXISIS-II is to be loaded, the monitor submits control to the start routine of the iRMX-80 Nucleus. After some internal initialization steps, iRMX-80 creates a number of ROM resident tasks; aside from the Terminal Handler and the Loader tasks, a task named RXIROM is activated which programs the Loader task to read the bulk of RXISIS-II from disk into memory. In addition, a small module is loaded from disk which contains the cursor positioning codes for the console terminal used. Loading these codes from disk rather than keeping them in ROM allows an easy adaptation of the computer system to terminals with potentially different control codes.

RXISIS-II is, in fact, a continuation of this task RXIROM. Similar to ISIS-II in the development system environment, RXISIS-II makes itself resident in part of the read-write memory. An extension of the operating system, a Command Line Interpreter, parses input entered at the console for valid commands, i.e., for the names of available disk files which hold executable programs, and loads these programs into the remaining free memory. Programs executed under ISIS-II or RXISIS-II are, in general, supposed to return control to the operating system when they are terminated; the Command Line Interpreter (but usually not the entire operating system) is loaded again, and the next program may be invoked and executed. The memory maps of ISIS-II and RXISIS-II are compared in Fig. 7; aside from the smaller size of the application programs area under RXISIS-II, and its slightly different boundaries (which were necessitated by the larger size of the iRMX-80 routines, compared to their ISIS-II counterparts, and by some constraints imposed by the hardware interfaces of iRMX-80), both systems are, indeed, very similar; virtually all well-behaved ISIS-II application programs can therefore be executed under RXISIS-II if they can put up with the lower "ceiling" of available memory. ("Well-behaved" means that the programs must route all their input and output operations over the standard ISIS-II functions, as opposed to direct access of peripheral devices.)

Although a wealth of auxiliary and utility programs can be executed under RXISIS-II (including, among others, a BASIC interpreter, and a full-screen text editor), this environment is not particularly suited for the execution of complex real-time controller programs like the CGCS. The emulation of ISIS-II imposes a code overhead which is not required if the proper iRMX-80 functions could be invoked directly as well.

Furthermore, RXISIS-II has to provide all functions of ISIS-II some of which are not required at all in a controller program. RXISIS-II was therefore designed to be replaced eventually by an arbitrary real-time system (as opposed to "program") which could be tailored to comprise exactly the required operating system routines. Such systems are, for example, a second, iRMX-80-based, version of the BASIC interpreter, and, of course, the CGCS. From the operator's point of view, there is no difference between loading a program under RXISIS-II, or an entire system; both are invoked by name. If the name of a system has been entered, however, a small module only is loaded by RXISIS-II which deposits the system's name in memory and re-starts iRMX-80. In due course, RXIROM loads the real-time system instead of RXISIS-II. (The cursor positioning codes for the CRT terminal are loaded in any case.) This task may or may not be kept active within the loaded system; in the CGCS, it is continued as the Command Interpreter task, which allows to utilize its resources which otherwise would have been wasted. Almost the entire read-write memory above the small data area of the ROM-resident system is available to the loaded real-time system.

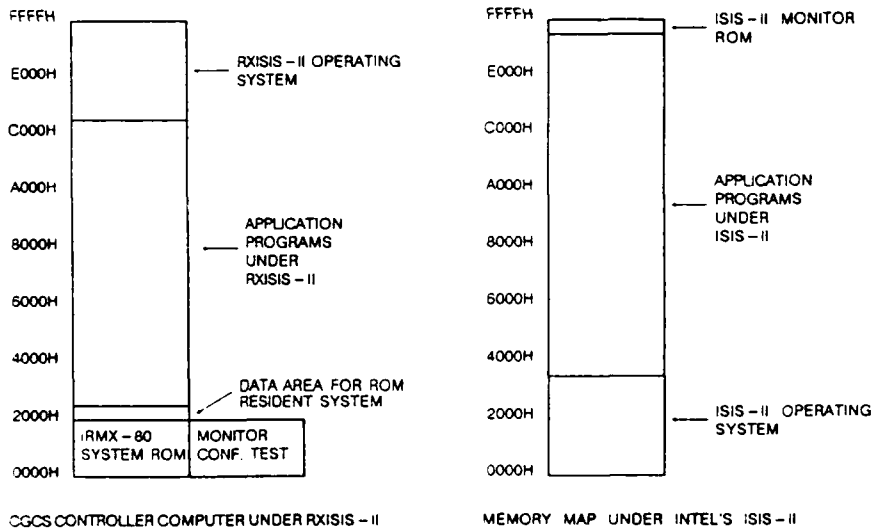


Fig. 7: Memory maps of the CGCS controller computer under RXISIS-II (a), and of an Intel development system under ISIS-II (b).

The Monitor may still be invoked from RXISIS-II or from any real-time system, either via the RXISIS-II DEBUG command, by pressing the "Break" key on the console terminal, or with the "Interrupt" switch on the cardcage. Since either the Monitor or iRMX-80 may be active, any access to the Monitor disables all iRMX-80 functions; the real-time system is virtually "asleep". This does not matter very much in the case of RXISIS-II, where fatal (disk) errors are also trapped by the Monitor, but it might be disastrous during the execution of a process control program. Therefore, entry to the Monitor was made more difficult in the CGCS by disabling the "Break" key detection; in case of a disk error, however fatal it may be under ISIS-II or RXISIS-II, control is not vectored to the Monitor either. (The "Interrupt" switch on the computer cardcage still permits access to the Monitor even while the CGCS is active, which is sometimes required for debugging purposes.) The Monitor may be used to inspect and modify memory locations and processor registers, and to set breakpoints in program code and execute programs until a breakpoint is encountered; iRMX-80 resumes full operation while a program is being executed from the Monitor. The Monitor also allows to re-boot RXISIS-II, which is, incidentally, the only way (short of a hardware reset) to terminate iRMX-80 BASIC, and it permits to activate the Confidence Test.

3.3 ROM Resident Software

3.3.1 The RXISIS-II Monitor

The RXISIS-II Monitor is kept in bank-switched ROM; it does therefore not consume any of the iRMX-80 system's resources. In order to permit proper memory bank switching, the Monitor must only be entered and left via special code sequences. (The same applies to the Confidence Test which is entered anyhow upon reset or from the Monitor only.)

The Monitor can be accessed in the following ways:

(1) After a system reset, or after execution of the Confidence Test.

(2) From RXISIS-II with the DEBUG switch.

(3) From RXISIS-II or any application system via a Break entered on the system console.

(4) From RXISIS-II or any application system via an RST 5.5 hardware interrupt which is generated by pressing the "Interrupt" switch on the cardcage.

(5) From RXISIS-II upon fatal disk errors.

Entry to the Monitor via a Break or an RST 5.5 interrupt is locked out during program file loading operations. (This is necessary as the Monitor uses the Loader buffer as scratch memory. Programs loaded while a Monitor interrupt happens might therefore be mutilated.) RST 5.5 interrupts are serviced immediately; entry requests given with a Break command become effective only after the Break key was released. Since a Break is initially noticed by the Terminal Handler as a transmission error, a beep and a "<" error character may be output by the Terminal Handler before the Break is serviced (compare chapter 3.3.4.2). This constitutes no actual error and can be ignored.

Note: Debugging in a real-time environment requires utmost caution! From the point of view of iRMX-80, the Monitor belongs to the task during whose execution it was invoked. By no means, the user must attempt to commence the execution of code belonging to a different task since this inevitably messes up the system totally. This demand is implicitly fulfilled if the user refrains from specifying a start address with the "G" (Go) command.

3.3.1.1 Monitor Commands

All Monitor commands may be entered with upper- or lowercase characters. They consist of one single character, and one or more parameters, if applicable. Multiple parameters must be separated either by a comma (",") or by a space; no space is permitted between the command character and the first parameter. All numeric parameters are interpreted as hexadecimal numbers; only their last two or four digits (depending on whether a byte or a word parameter is required) are relevant. Errors during parameter entry can therefore be corrected by repeating the parameter without an intervening delimiter until the last two or four places are correct. Input lines are generally entered with "Return"; any other input than hexadecimal numeric characters ("0" through "9" and "A" through "F"), comma, space, and Return causes a "COMMAND ERROR" message. A period (".") is used by the Monitor as an input prompt character. The command syntax was chosen similar to the ISIS-II Monitor.

The following Monitor commands are available:

D<locfrom>[,<locto>]
Display Memory Contents:

The contents of all memory locations between and including <locfrom> and <locto> are displayed in hexadecimal. Only the contents of <locfrom> are displayed if <locto> is omitted or less than or equal to <locfrom>.

E Exit to the Current System and Close Open Files:

Upon this command, the Monitor is left, and control is submitted to the currently active operating system. In general, it lies in the responsibility of the currently active system to close all open disk files, and to restore a defined state of all routines used (compare chapter 3.4). This function requires therefore the "cooperation" of the current operating system, and it is therefore not available with some application systems (e.g., with iRMX-80 BASIC). In such a case, the Monitor tries to execute the "Q" (Quit) command. Either command must be explicitly confirmed by the user.

F<locfrom>,<locto>,<byte>
Fill Memory With a Specified Byte:

The locations between and including <locfrom> and <locto> are overwritten with <byte>. An error message is output if the operation would extend into ROM.

G[<start>][,<breakpt1>[,<breakpt2>]]
Go - Execute Program Code:

The Go command permits the execution of program code. The execution begins at the location <start> or, if no <start> address was specified with the command, at the location determined by the current Program Counter contents (compare "X" command). Up to two breakpoint locations may be specified at whose execution the program is to be interrupted. These locations must lie in RAM, and they must contain the first byte of an executable machine code instruction. Furthermore, they must not be overwritten by an intervening program loading operation lest the breakpoint becomes ineffective and unpredictable results ensue after the monitor was activated the next time. In a real-time environment, the breakpoints may belong to arbitrary tasks. The start location, however, must belong to the task during whose execution the Monitor was invoked. Note that breakpoint specifications must be preceded by a delimiter (comma or space) if the default start location is to be used.

H<param>,<param>
Hexadecimal Addition and Subtraction:

The "H" command performs an addition and subtraction of two two-byte parameters. The sum and the difference (in 2's complement notation) of the two parameters are displayed.

I[<port>]
Input Data From I/O Port:

This command allows to read the contents of any arbitrary I/O port. <port> is interpreted as a one-byte number in hexadecimal notation. For multiple inputs from the same port, the <port> parameter may be omitted. The command defaults to the last port specified in this case.

M<locfrom>,<locto>,<destloc>
Move Memory Contents:

The "M" command permits to move the memory contents between and including <locfrom> and <locto> to locations starting with <destloc>. <destloc> should not lie between <locfrom> and <locto> in order to prevent the modification (by partial repetition) of the byte pattern to be moved. <destloc> must be located in RAM. Memory contents are transferred in increasing address order.

O[<port>],<data>
Output Data Byte To I/O Port:

The data byte specified with the command is output to the I/O port specified with <port>. The <port> address is interpreted as a single-byte number and may optionally be omitted; in this case, the Monitor defaults to the last output port specified. The <data> byte must be preceded by a delimiter if <port> is omitted.

P{0|1} Printer Output On/Off:

All Monitor output shown on the console CRT can be routed, in addition, to the printer. A "P1" command turns on this function, a "P0" command turns it off. A built-in time-out function disables the printer output if the printer did not respond within a certain time (10 to 30 seconds, depending on the clock frequency used). (This may also happen during lengthy Monitor output operations with the "D" command if the printer's input buffer needs longer than the time-out to be emptied. If this is regarded as a problem, it can be fixed by reducing the printer buffer size.)

Q Quit the Monitor and Re-Boot RXISIS-II:

The "Q" command permits to leave the Monitor and to re-boot RXISIS-II. Open disk files are not closed before RXISIS-II is re-booted; files which may have been open for writing or updating will be mutilated in this case. Great care is therefore required when the "Q" command is used to prevent the loss of output files. The command must therefore be confirmed explicitly by the user.

S<address>,<data>[,<data>[,<data>...]]
Substitute Memory Contents:

This command allows to change the contents of arbitrary locations in RAM. When the <address> input is terminated with the delimiter character, the current contents of the location at <address> are displayed. Hexadecimal data entered at this stage are used to replace the old contents of the specified location. No change of the location displayed is made if another delimiter is keyed in without an intervening hexadecimal number. In either case, the address is incremented by one, and the above procedure is repeated. The command is terminated with a "Return".

X Display Register Contents:

X<reg><data>[,<data>[,<data> ...]]
Modify Register Contents:

Two versions of the "X" command permit the display and the modification of register contents, respectively. Upon entry of a plain "X", the contents of all processor registers are displayed. The second command mode permits to modify the register contents, beginning with the register specified with <reg>, similar to the "S" monitor command. <reg> may be "A", "B", "C", "D", "E", "H", and "L" for the corresponding registers, "F" for the flags, and "S" and "P" for the stackpointer and the program counter, respectively. The current contents of the register to be modified are displayed; the register may either be overwritten or preserved, depending upon whether a hexadecimal number followed by a delimiter, or a delimiter only is entered. Following a data entry with a delimiter permits to modify the next register; the command is finally terminated with a "Return". (The sequence of registers is A, F, B, C, D, E, H, L, SP, and PC for either "X" command.)

Z Enter Confidence Test:

The Confidence Test routine can be entered with the "Z" command at any time. A user confirmation is required. Note: When the Confidence Test returns to the Monitor, it resets the stackpointer to the top of the Monitor's stack, and all other registers to zero. It is therefore not possible to continue the execution of a program after the execution of the Confidence Test.

3.3.1.2 Other Monitor Functions

The Monitor provides, in addition to the above utilities, detailed system error message output. All errors which are considered fatal under RXISIS-II are trapped there (essentially, these are the errors defined as fatal under ISIS-II, plus all errors happening during system bootloading). Application systems may use this feature as well; it is not utilized, though, by the CGCS. The calling sequence for the error message generation routines is outlined in chapter 3.3.6.

3.3.1.3 The Monitor in a Real-Time System

As mentioned above, the Monitor is executed as if it belonged to the task which was active while the Monitor was invoked. This is true although the interrupt dependent iRMX-80 functions are disabled while the Monitor is active. Due to the inherent complexity of a real-time system, great care is required to prevent system breakdowns when Monitor operations are performed. This does not only involve caution when register or memory contents are changed; this applies particularly to program execution, and to Monitor exiting.

In most cases, the "G" command, specified without start and breakpoint addresses, is the most straightforward way to return to real-time operations. The system continues where it was interrupted (unless it was interrupted by a fatal error under RXISIS-II, in which case RXISIS-II has to be re-booted anyhow), and behaves as if it never had been interrupted at all. (Since the Monitor halts iRMX-80, periodic operations, particularly, the timekeeping functions, are disabled temporarily. The time displayed by the CGCS will therefore differ from its correct value by the duration of the Monitor operation.)

The "E" command, in contrast, permits to terminate a program executed under RXISIS-II, and it also terminates the CGCS. Application systems which want to utilize this feature must take care not to override iRMX-80 task scheduling. Upon an "E" command, the Monitor executes a routine which must be provided by the application system; its start address has to be stored previously in a dedicated Monitor location in RAM (see Appendix 3). Control must by no means be passed directly to the system task which performs the termination operations like disk file maintenance; such an approach would mean that the task interrupted by the Monitor call might be continued as another task, which is an absolutely fatal error in any real-

time system. The only unambiguous procedure which allows a safe Exit operation is discussed in the next chapter.

It should be noted, though, that a Monitor Exit only closes open disk files but does not perform any of the other clean-up chores which are essentially required for a safe operation of a process control system like the CGCS, particularly if it is in charge of the puller. The "E" command should therefore used in case of an emergency only from within the CGCS.

The "Q" command, finally, is an uncompromising way to quit the Monitor: the current system is simply destroyed by restarting iRMX-80 and re-booting RXISIS-II. Still, it is the only way to leave systems such as iRMX-80 BASIC which do not provide any other means for the termination of their services. (The "Q" command processing sequence is automatically entered by the Monitor if an "E" command was issued but no provisions were made for exiting the real-time system with a preceding clean-up.)

A Hardware System Reset is approximately equivalent to entering the Monitor, e.g., via a Break, and quitting with the "Q" command. Upon a Reset, the Monitor is entered via its Restart sequence; entering a "Return" permits to re-boot RXISIS-II.

3.3.1.4 Exit From the Monitor

Due to memory bank switching, the Monitor and Confidence Test routines may only be left via two paths, namely:

(1) Via a RST 1 instruction which bootloads RXISIS-II, and

(2) Via a jump to an exit sequence located at 1FF8H.

The first path is used by the Monitor upon a Quit ("Q") command, the second, upon Go ("G") and Exit ("E") commands.

Systems which ought to provide the possibility of an Exit ("E") Monitor command have to pursue the following steps:

(1) They have to store the start address of an Exit Routine in the Exit Pointer locations of the Monitor, i.e., at the addresses 2008H (low byte) and 2009H (high byte) (see Appendix 3).

(2) This Exit Routine must be written as a subroutine (which is eventually called by the Monitor) which must return to the Monitor (with a RETURN instruction) in any case. The

Exit Routine may set a flag which is polled by a system task in charge of the file handling. Whenever the system task finds this flag set it should close all open files and perform subsequently all actions considered appropriate. The Exit Routine should not attempt to directly accomplish these operations itself. This approach was chosen for RXISIS-II where the exit flag is checked each time the ISIS-II entry point or one of the ISIS-II Monitor subroutines is invoked. There may be a considerable delay between exiting the Monitor and a call to any RXISIS-II system routine, though, during which the program to be terminated keeps on running.

A different (and faster) approach is employed by the special Disk I/O routines (compare chapter 4.2.3) which are used by the CGCS: The Exit Routine sends, in this case, a message to the Disk I/O Interface task which triggers the file closing sequence. A message can be sent from any task, and the termination sequence is started immediately; hence, no noticeable delay between an "E" command and its execution can be seen in the CGCS.

3.3.2 The RXISIS-II Confidence Test

The RXISIS-II Confidence Test code is kept in bank-switched ROM and does therefore not consume system memory under regular iRMX-80 operation. It can be invoked and executed from the Monitor (compare chapter 3.3.1). In addition, the Memory Test sequence is run after each power-up reset in order to confirm proper system operation. It is not executed upon a reset triggered when the is already running, which permits to inspect memory after a system reset necessitated, e.g., by a software failure.

All functions of the Confidence Test can be selected interactively in turn. Ample user information is provided.

Upon completion, the Confidence Test returns control to the Monitor.

3.3.2.1 Memory Test

The Memory Test comprises the following features:

(a) Verification of the ROM checksums.

(b) Check of the proper function of the RAM.

(c) Initialization of all RAM locations (except between 3000H and 30FFH) with zeros.

The current status of the Memory Test routine is output on the console CRT by means of eight binary digits which represent the address range currently under test. The routine halts if an error is detected; the display on the console CRT indicates in this case the whereabouts of the erroneous memory location. The following sequence of operations and output is performed:

CRT DISPLAY	
[NONE]	RAM check at locations 2000H - 2020H.
0000 0000	ROM sequence test (ROM 0 - 1 - 2 - 3).
0000 1111	Checksum test of Bank 0, ROM 0 and 1.
0001 1111	Checksum test of Bank 0, ROM 2 and 3.
>BEEP<	
0000 0000	Preparation for the RAM test.
XXXX XXXX	The display counts down twice from 1111 1111 to 0010 0000. The binary numbers displayed indicate the high byte of the address under test.
>BEEP<	
0000 1111	Checksum test of Bank 1, ROM 0 and 1.
0001 1111	Checksum test of Bank 1, ROM 2 and 3.
>BEEP<	

3.3.2.2 CRT Console Test

The CRT Console Test permits to check the communication interface to the console terminal. Each character entered at the console is echoed multiplely in order to fill 24 lines with 80 characters each. Control characters are indicated by an up-arrow ("^") preceding the pertinent regular ASCII character. Since there are no carriage return or line feed characters embedded between the 80*24 characters, a bottom screen line which is not completely filled indicates problems with the transmission protocol (compare chapter 2.3.8). Transmission errors such as parity, overrun, and framing errors detected in the data received by the system are reported. The Console Test can be left at any time by entering a space.

3.3.2.3 Printer Test

The Printer Test routes any input from the console to the line printer. The console input is transmitted literally, i.e., it

is necessary to enter a carriage return and a line feed in order to receive these characters on the printer. All control characters including "Escape" are transmitted. The test can be terminated by entering two consecutive "Escapes".

3.3.2.4 I/O Port Test

The I/O Port Test sequence permits to read data from and to write to any arbitrary I/O port. Address and data inputs are requested and output is given in hexadecimal notation. (The "I" and "O" commands of the Monitor are probably more convenient for this purpose, though.)

3.3.2.5 Floppy Disk Test

The Floppy Disk Test provided with the Confidence Test routines supports two standard size, single density, single side drives under an iSBC 204 Disk Controller. The test performs the following operations, first on drive 0, then on drive 1:

(1) Recalibrate (position head over track 0).

(2) Format (provide track and sector information on the disk). Disks formatted with this function are not compatible with ISIS-II or RXISIS-II!

(3) Verify CRC (check the Cyclic Redundancy Check checksums generated during formatting).

(4) Random Read/Write Test (write data into randomly distributed sectors and read them back for verification).

NOTE: The contents of the disks used in this test are irreversibly destroyed!

3.3.3 The iRMX-80 Nucleus

The iRMX-80 Nucleus resides in ROM Bank 1 which is to remain active during all regular iRMX-80 based operations. The ROM resident iRMX-80 routines comprise the necessary iRMX-80 system initialization information, and the iRMX-80 Nucleus proper, i.e., the routines responsible for the maintenance of the operating system, for the proper scheduling of tasks, and for the transfer of messages between them.

After the internal iRMX-80 structures were created, a number of iRMX-80-supplied tasks or their replacements (the Alternative Terminal Handler and the Loader Tasks), and a task named RXIROM start running. RXIROM programs the Loader Task to read the cursor positioning routine RXISIS.PSC from disk (compare chapter 3.3.4.1.7), and to subsequently load either RXISIS-II (from a file RXISIS.BIN), or any other real-time application (like iRMX-80 BASIC, or the CGCS), and vectors control to the loaded code. Since the loaded systems have to refer heavily to addresses in ROM (e.g., for all Nucleus function calls), it is essential that the program loaded from disk was actually configured for the ROM version used. This is checked by RXIROM by means of a ROM version code which must be provided by any loaded system in a particular memory location (compare Appendix 3).

3.3.4 The Alternative Terminal Handler

The Alternative Terminal Handler replaces the Full Terminal Handler of iRMX-80. All functions performed by the iRMX-80 Terminal Handler are identically available from the Alternative Terminal Handler. The following major differences between the iRMX-80 Full Terminal Handler and the Alternative Terminal Handler apply:

* Improved line-editing.

* Fixed-screen CRT console output possible.

* Additional printer output supported.

* Additional single character input feature.

* Additional control functions.

* Break detection.

The most important feature of the Alternative Terminal Handler is, aside from more convenient line-editing, the possibility of a Fixed Screen output, which encompasses the output of a dedicated cursor positioning code prior to each output action. After each regular output operation, the Terminal Handler re-positions the cursor to the current end of the input line, whose position on the screen can be freely specified by the programmer. The CRT terminal used must permit direct cursor addressing.

At system restart, the Terminal Handler is in the conventional scrolled output mode, i.e., input data is always echoed in the currently last line on the CRT screen (which is, due to the automatic scrolling of a CRT terminal, in most cases the bottom line of the screen). Output data is also appended sequentially in the currently last line. The CRT screen thus represents a sequential protocol of the most recent I/O operations. This approach is no more suitable for a genuine real-time system: Miscellaneous output is usually generated in a random sequence and at different rates for different items. Since the occurrence of an output operation may not be predictable, it is close to impossible to permit the entry of operator input without disturbance by interspersed output. The only approach to avoid this problem and to permit the operator to monitor a complete overview of the system's most recent output is to use a Fixed Screen approach where each input or output item has its dedicated place on the CRT screen. Updating of the CRT screen does therefore no more affect the location of a certain item on the screen, quite in contrast to the scrolled mode.

In the Fixed Screen Mode, the Alternative Terminal Handler reserves two contiguous lines where the input echo is built. No output data should be directed to these lines. While it lies in the responsibility of the application software to provide the proper cursor positioning code in front of each output string, the Terminal Handler re-positions the cursor automatically to the current end of the input echo string after each output action. No interference between output and input operations can thus happen since all input characters echoed are simply appended to the input line. The input line itself is cleared by the Terminal Handler when a new input string is requested; the position of the input line on the screen and, if required, an input prompt string may be specified by the programmer.

Similar to the Full Terminal Handler, the Alternative Terminal Handler provides a type-ahead feature, i.e., data can be entered although no input request from other tasks is currently pending. The type-ahead buffer permits the entry of up to 80 characters, depending on the number of type-ahead lines (up to 20 (empty) lines can be entered into the type-ahead buffer).

Two I/O features are new, compared to the iRMX-80 Terminal Handlers: First, output to a printer (or any other device which can be connected to a serial output port and which receives output but does not generate input), and, second, a single character input (in contrast to the line-oriented input featured by iRMX-80). Furthermore, the list of Terminal Handler Control Characters (RQCTAB) was extended, and hence the

number of control features. In addition, an arbitrary routine supplied by the application system (R@@BRK) is invoked if a "Break" is received from the console, and if the break detection is enabled.

3.3.4.1 Programming Interface

Both with regard to the programming and to the operator interface, the functions of the Alternative Terminal Handler are upwardly compatible to those of the iRMX-80 Full Terminal Handler. The following information is therefore kept concise as far as it is identical to the programming of the iRMX-80 Full Terminal Handler.

3.3.4.1.1 Line Input Operations

Requests for an input line can be directed to either the Regular Input Exchange RQINPX or to the Debug Input Exchange RQDBUG. During regular operation, RQDBUG is inactive; it becomes active only after a Cntl-C was entered at the console.

The format of the Read Request Message which has to be sent either to RQINPX or to RQDBUG is identical to that of the iRMX-80 Terminal Handler; the following TYPE values are permitted:

* READ$TYPE (8), which permits reading using the type-ahead feature of the Alternative Terminal Handler.

* CLRRDTYPE (9), which clears the type-ahead buffer prior to requesting an input line.

* LASTRDTYPE (10), which disables the input from RQDBUG and enables input via RQINPX. LASTRDTYPE preserves the contents of the type-ahead buffer.

3.3.4.1.2 Console Output

Two exchanges are available for generating console output, namely RQOUTX and RQALRM. RQOUTX may be disabled, which is not possible for RQALRM. As with the iRMX-80 Terminal Handler, the following message types are permitted:

* WRITE$TYPE (12): A message of this type may be sent to either exchange.

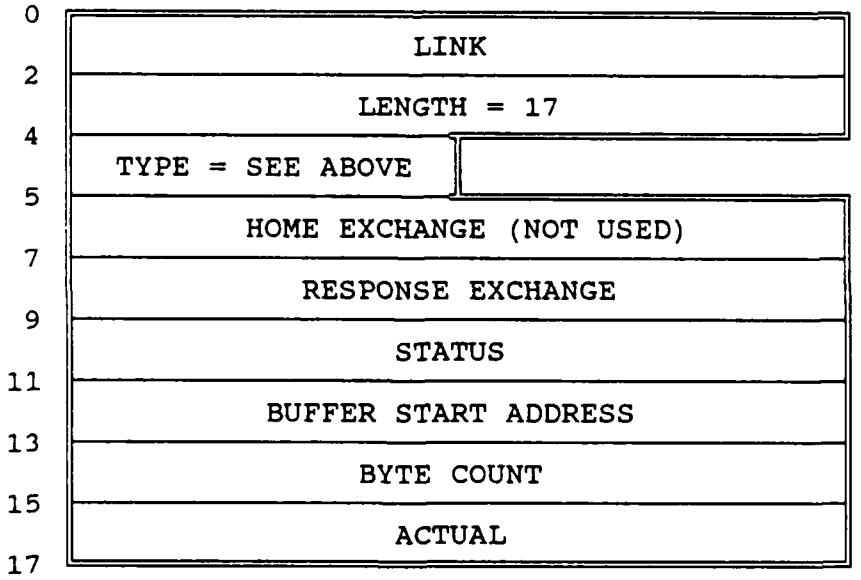
* ALARM$TYPE (11): Messages of this type must only be sent to RQALRM. The output of the bytes specified with the message is initialized with a string of five asterisks ("*") and two BEL characters. Note: In Fixed Screen output mode, the output strings supplied by the user tasks must be initiated with a cursor positioning code sequence. Since this sequence is only output after the above alarm string, the position of this string on the screen will be undefined. The use of ALARM$TYPE is therefore discouraged if in Fixed Screen mode.

3.3.4.1.3 Printer Output

All output which should be sent to the printer (or whatever serial output device is connected to the second RS-232 port of the system) must be sent to the exchange RQPRNT. Only WRITE$-TYPE (12) is permitted as message type.

3.3.4.1.4 Line Input and Output Request Messages

The format of the request messages for the three above operations is identical.

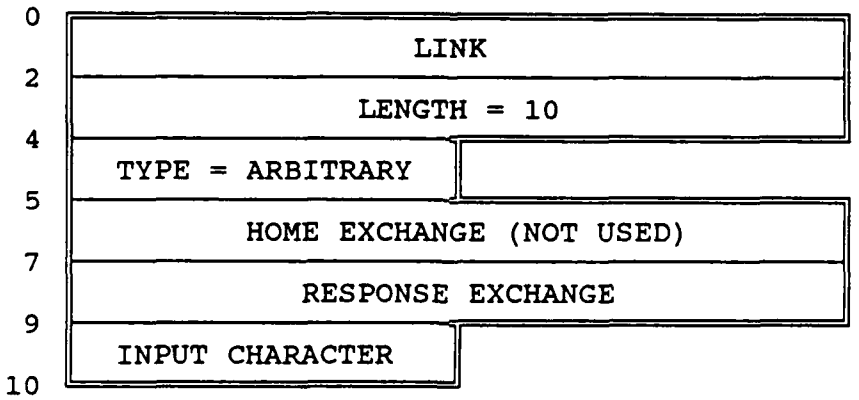


STATUS and ACTUAL are set by the Alternative Terminal Handler; all other items must be provided by the programmer. STATUS is either 0 if the requested operation was performed properly, or 18 (BAD$COMMAND) if an illegal TYPE parameter was specified. ACTUAL is set to the number of bytes actually input or output.

3.3.4.1.5 Single Character Input

Single character input as provided by the Alternative Terminal Handler bypasses the control character evaluation and line editing functions of the Terminal Handler; it was included for compatibility with ISIS-II and its Console Input (CI) routine. Input of a single character can be requested by sending a request message to the exchange RQCHIX. The next character input after the request message was received at RQCHIX is returned with the request message rather than being processed by the Terminal Handler. An application system which chooses to utilize this function should make sure that always at least one request message is waiting at RQCHIX in order to prevent input characters which were entered while no request message was waiting from being added to the Terminal Handler's buffer and being therefore lost for the routine using the single character input. Furthermore, line input request messages of type CLRRDTYPE (9) should be used for intervening and concluding line input actions in order to clear spurious contents of the type-ahead buffer.

A request message sent to RQCHIX must have the following structure:



INPUT CHARACTER is returned by the Alternative Terminal Handler; in order to conserve time, no further syntax check is performed on the request message.

3.3.4.1.6 Output Mode Setup and Input Prompt String Selection

The subroutine RQISCM was provided to permit the selection of the output mode (scrolled output or Fixed Screen), of the input lines in Fixed Screen mode, and of an input prompt string. The maximum length of this string is determined by the relation

```
string length = 32 - cursor positioning string length -
              - 2*(line clearing code string length +1)
```

Longer strings are truncated.

An output mode change can be effected at any time by a call to the Terminal Handler subroutine RQISCM. This routine should be called by only one task within the application system, preferably while no input request is pending. The following parameters must be passed to RQISCM:

CALL FROM PLM:

```
CALL RQISCM (.inline,.printl,.inistr,inisl)
```

CALL FROM FORTRAN:

```
CALL RQISCM (inline,printl,inistr)
```

with:

inline	Number of the line on the CRT screen reserved for the input echo. inline = 0 ... Conventional Scrolled Mode inline <> 0 .. Fixed Screen Mode
printl	Number of printable characters within the input line initialization string (must be less than or equal to the initialization string length)
inistr	Input line initialization string: Must contain all information exceeding the cursor positioning and line clearing codes; input prompt characters may be entered here.
inisl	Input line initialization string length.

PARAMETERS FOR ASSEMBLY LANGUAGE CALLS:

```
STACK Input Line Number Storage Location
STACK Printable String Length Storage Location
B+C . Initialization String Start Address
E ... Initialization String Length
```

Note that in Fixed Screen mode the Alternative Terminal Handler clears the input line specified with the "inline" parameter, and the line following it. No check for the validity of the line number submitted is performed. The two input lines may virtually be located everywhere on the CRT screen; however, the bottom line of the screen should not be included into the input area if the input echo (including a leading prompt string) might exceed one line on the screen; otherwise, the display would scroll up when the input line is terminated with Carriage-Return.

3.3.4.1.7 Cursor Control Code Generation

In general, the cursor positioning and line clearing codes required depend on the type and make of the CRT terminal used. In general, they must be determined at system configuration time. In the implementation of the Alternative Terminal Handler on the CGCS computer, the terminal-dependent codes are kept in a disk file which is loaded in front of any real-time application system. This disk file has to provide the following labels at the specified addresses:

27D0H ... Vector to the Cursor Positioning Code Generation Routine.
27D3H ... Vector to the Line Clearing Code Generation Routine.

27D6H ... Cursor Up Code.
27D8H ... Cursor Down Code.
27DAH ... Cursor Left Code.
27DCH ... Cursor Right Code.
27DEH ... Cursor Home Code.

27E0H ... Clear Screen Code.
27E2H ... Clear Line Code.

The locations from 27E4H through 27FFH can be used for the Code Generation routines. Two bytes are reserved for each simple code; the chronologically first byte of the code must be kept in the high byte of the code word, the second, in the low byte. (This can readily be achieved using a DW Assembly Language directive, followed by the two codes in chronological order.)

The layout of the operating system environment on the CGCS computer is designed to accommodate cursor positioning routines for terminals with control codes not exceeding four characters for positioning and two characters for line clear-

ing in the address space from 27D0H through 27FFH. Although it is possible to provide longer cursor positioning code generation routines under RXISIS-II, this is not feasible for the CGCS. Alternative cursor positioning routines must therefore be kept in the memory range mentioned above in order to avoid collisions with the CGCS.

With regard to speed requirements, the parameter passing conventions of either PL/M or Fortran could not be maintained for the Code Generation routines. They must therefore be written in Assembly Language. The following parameters are required:

D ... Line number (≥1) (I) *)
E ... Column number (≥1) (I) *)
H+L . Pointer within the output string (I,O)
A ... Length of the positioning string (O) *)

Line and column numbers are required for the generation of the Cursor Positioning Code only.

3.3.4.1.8 Break Detection

The Alternative Terminal Handler monitors the Break status of the console I/O line. A routine R@@BRK (which is provided by the ROM resident code of RXIROM) is invoked whenever a break was detected (or, more accurately, after the break condition was terminated), provided an enable flag (RQENBK) which is declared PUBLIC by the Terminal Handler was set (to 0FFH).

3.3.4.1.9 Public Parameters

In addition to the entry exchanges, the following parameters are declared PUBLIC by the Alternative Terminal Handler. They may be accessed by user routines but should be handled with great care. Some of them must by no means be changed by external routines. The PUBLIC parameters are listed below; the routine where they are declared PUBLIC is also noted.

RQDBEN (RQTHDI) Debug Enable Flag:
0 ... Cntl-C and Cntl-A are disabled.
0FFH .. Cntl-C and Cntl-A active.

"Cntl-C" and "Cntl-A" can be locked out if the Alternative Terminal Handler flag location RQDBEN (debug enable) is

reset to a zero value. A non-zero value leaves them active but inoperative if no task waits at RQDBUG.

RQDBMD (RQTHDI) Debug Mode Flag: *)
0 ... Input via RQDBUG.
0FFH .. Input via RQINPX.

The Debug Mode flag is set by the input task of the Alternative Terminal Handler. It must not be modified by external code but may be used to monitor the current input mode (Regular or Debug Input).

RQENBK (RQTHDI) Enable Break Detection Flag:
0 ... Break detection disabled.
0FFH .. Break detection enabled.

The routine R@@BRK is invoked if this flag is set and a Break condition is detected. Otherwise, the Break is ignored.

RQTHMC (RQTHDI) Input Mode Changed Flag:
0 ... No Input Mode change.
0FFH .. Input Mode changed from Debug to Regular.

This flag is supplied as an indicator for operation mode changes of the Input Terminal Handler. It is set when RQTHDI cnanged back from Debug Mode (i.e., input via RQDBUG) to Regular Mode (i.e., input via RQINPX). Application tasks may monitor this flag in order to determine whether a restoration of a Fixed Screen output is necessary (which is probably the case if a background task was accessed via RQDBUG). The flag RQTHMC has to be reset by the user code. Its status has no effect on the operation of the Alternative Terminal Handler.

R@ECHO (RQTHDO) Echo output entry exchange *)

No messages should be sent to this exchange by user tasks!

```
R@@TDS (RQTHDO) Terminal Output Task Descriptor *)

R@@PDS (RQTHDO) Printer Output Task Descriptor *)
```

These two PUBLIC variables are the start addresses of the Task Descriptors of the two tasks which interface the Console (RQOUTX) and Printer (RQPRNT) Output Request exchanges to the output task (RQTHDO) proper. Suspending either of them locks out output via the corresponding exchange. The two tasks are suspended and resumed by the input of Cntl-S and Cntl-Q (for console output) and of Cntl-E and Cntl-F (for printer output) on the console, respectively.

```
R@OENA (RQTHDO) Terminal Output Enable Flag *)
                    0 ... Output disabled.
                  0FFH .. Output enabled.

R@PENA (RQTHDO) Printer Output Enable Flag *)
                    0 ... Output disabled.
                  0FFH .. Output enabled.
```

These two flags permit to enable and disable terminal and printer output (via RQOUTX and RQPRNT), respectively. They should be modified externally only with great care since they must be used in conjunction with enabling and disabling the interface tasks (see above). Disabling is, in general, done by resetting the appropriate flag, enabling, by resuming the interface task after the flag was set. Direct suspending and resuming of the interface tasks is discouraged.

```
R@OKIL (RQTHDO) Terminal Output Kill Flag.
                    0 ... Regular output.
                  0FFH .. Output discarded.

R@PKIL (RQTHDO) Printer Output Kill Flag.
                    0 ... Regular output.
                  0FFH .. Output discarded.
```

The two Output Kill flags may be set or reset at any time. Output requests directed to RQOUTX and RQPRNT, respectively, are simply ignored if they are set. They do not affect the system otherwise. The setting of the Output Kill flags is toggled upon each Cntl-O or Cntl-V input.

R@INLN (RQTHDO) Input line number *)
0 ... Scrolled output.
<>0 ... Input line number in Fixed Screen mode.

The input line number/flag held in this variable may be read only by external routines.

*) Do not attempt to modify these locations!

3.3.4.2 User Interface of the Alternative Terminal Handler

The line input feature of the Alternative Terminal Handler has been improved significantly in user-friendliness, compared to the iRMX-80 Terminal Handlers. In general, the input line echo on the console CRT screen corresponds exactly to the contents of the input buffer. A line entered on the console is submitted to the system only after the line was terminated with one of four Line Termination Codes. Until that happened, editing is possible via two Editing Codes.

The length of an input line is limited to 80 characters, i.e., to the width of a CRT screen. Due to input prompt characters or strings, the line may, however, extend over two lines on the console screen. (The console terminal should be set to an Auto New Line Enable mode if possible.)

A beep is output if the user tries to enter data beyond the available buffer length. Type-ahead is available for up to 80 characters which may be entered before an input line is requested by the system. Since the type-ahead input is echoed on the CRT screen only when it is requested, it has to be entered blindly. Unless a type-ahead line was terminated with a Line Termination Code, it can be edited arbitrarily, either before or after it was echoed. Note that Cntl-X deletes the type-ahead buffer completely, i.e., even type-ahead lines which were already terminated. A warning beep is output if the operator tries to exceed the type-ahead buffer length.

In Fixed Screen Mode, the input line is displayed at a fixed location on the screen. Its display area has therefore to be cleared periodically. This is done by the Alternative Terminal Handler after an input request message was received. The previous input line is thus displayed even after it was entered and passed on to the system by the Alternative Terminal Handler; the cursor is, however, moved to the extreme left end of the input line. This behavior was required in

order to permit complete input lines which were already terminated to be displayed in type-ahead mode. After an input request was received, the input area is cleared, the input prompt string is output if applicable, and the cursor is moved to the first location on the screen available for the input line echo unless there was already input in the type-ahead buffer, which is otherwise written to the screen with the cursor positioned after its end.

The Alternative Terminal Handler accepts 16 control characters with a special meaning. Any such character which is input on the Console Terminal (and which is not caught by a single character input request beforehand) triggers the appropriate action listed below. Other control characters are rejected unless they are preceded by Cntl-P.

(1) Line Termination Codes:

The following codes terminate an input line and advance the input buffer to the routine requesting input. In general, the termination characters are appended to the input data unless there is no more enough room in the buffer.

CR Carriage Return: Converted to a CR-LF pair and written to the buffer and echoed as CR-LF.

LF Line Feed: Treated identically to Carriage Return.

ESC Escape: Appended to the buffer, echoed as "$"+ CR-LF (no "$" if entered in type-ahead mode). (Note that Escape is used as an input line clearing command within the CGCS.)

Cntl-Z Control-Z: Deletes all buffer contents, transmits an empty buffer. Echoed as a CR-LF pair. In type-ahead mode, Cntl-Z may be used to delete the last, yet unterminated, line entered, without affecting the contents of preceding input lines. (Due to system timing problems which could be overcome only with a great expenditure of code and/or processing time, an input line entered into the type-ahead buffer immediately after one or more Cntl-Z characters may not be echoed although it is regularly advanced to the task requesting input.) Control-Z is interpreted as a program termination code by some auxiliary programs (e.g., the Macro Command Editor) running under RXISIS-II.

(2) Line Editing Codes:

RO Rubout: Deletes the last character in the input buffer and on the screen. (It is, however, impossible to remove a character in the last column of the CRT screen from the display if the terminal used does not permit a "scroll-back" of the cursor from the leftmost position of a line into the last position of the preceding line. Nevertheless, the erased character is removed from the input buffer; a correct display can be obtained with Cntl-R.)

Cntl-X Control-X: Deletes the buffer and the type-ahead buffer completely. Appends a "#" to the input line echo and advances to the next line on the screen (not in type-ahead mode).

(3) Miscellaneous Control Codes:

Cntl-P Control-P: The character following Cntl-P is input literally even if it is a control character.

Cntl-R Control-R: Restores the input line echo on the console CRT. No visible effect if input line does not extend over two physical CRT screen lines. Can be used if characters were deleted in an input line extending over two CRT lines and the cursor did not move up to the upper echo line.

Cntl-S Control-S: Suspends regular console output. The tasks requesting regular output are halted. Does not affect output sent to the RQALRM exchange. Suspending output already suspended has no effect.

Cntl-Q Control-Q: Resumes output suspended with Cntl-S. Resuming output not suspended has no effect.

Cntl-O Control-O: Regular output is deleted if Cntl-O was entered. The routines or tasks generating output keep running; their output is lost. Regular output can be restored if Cntl-O is entered a second time.

Cntl-E Control-E: Suspends printer output. Tasks requesting printer output are halted. Suspending printer output which is already suspended has no effect.

Cntl-F Control-F: Resumes printer output suspended with Cntl-E. Resuming output not suspended has no effect.

Cntl-V Control-V: Printer output is deleted if Cntl-V was entered. Tasks requesting printer output keep running; their output is lost. Printer output can be resumed if Cntl-V is entered a second time.

Cntl-C Control-C: This control is only effective if the Debug Enable Flag RQDBEN is set, if the regular input exchange RQINPX is active, and if a request message waits at RQDBUG. In this case, all console input is directed to the request messages waiting at RQDBUG, and RQINPX is no more serviced. In addition, a message is sent to the exchange RQWAKE unless there is a message already waiting there. The regular input mode is restored and the Mode Changed Flag RQTHMC set to 0FFH if a message of LASTRDTYPE (10) is sent to RQDBUG. The type-ahead buffer is cleared. (Control-C is used by the iRMX-80 Debugger, and by iRMX-80 BASIC. It has no effect whatsoever in the CGCS.)

Cntl-A Control-A: Cancels the effect of Cntl-C; all input is directed to the regular input exchange RQINPX. Cntl-A is only active if RQDBEN is set and RQINPX was not serviced (i.e., in Debug mode). The Mode Changed Flag RQTHMC is set to 0FFH, and the type-ahead buffer is cleared. (Control-A is used in conjunction with the iRMX-80 Debugger only; it has no effect whatsoever in the CGCS.)

The Terminal Handler indicates console input transmission errors by echoing special characters, accompanied by a beep. The erroneous character is discarded, and the following characters are echoed:

	FRAMING ERROR	OVERRUN ERROR	PARITY ERROR
@	-	-	+
?	-	+	-
>	-	+	+
=	+	-	-
<	+	-	+
;	+	+	-
:	+	+	+

3.3.5 The Generic Loader Task

The Generic Loader Task RQLOAD is designed for a twofold application:

(a) as a Bootloader to load entire iRMX-80 based systems into RAM, and

(b) as a standard Loader during the regular operation of an iRMX-80 system.

The Generic Loader permits, in contrast to the iRMX-80 Bootloader, to specify a device and filename at runtime, which allows to use it as a standard Loader. Similar to the iRMX-80 Bootloader, and unlike the iRMX-80 standard Loader task LOAD, it does not require the support by the Directory Services task DIRSVC which keeps its code sufficiently short to make it fit into a Bootloader module.

Several functions were provided in addition to those supported by the iRMX-80 task LOAD:

* It is possible to protect memory areas by setting low and high boundaries for the Loader's operations. Any attempt to load code into locations with an address less than the low boundary or greater than the high boundary terminates the loading operation and sets the STATUS field of the request message to a corresponding error value.

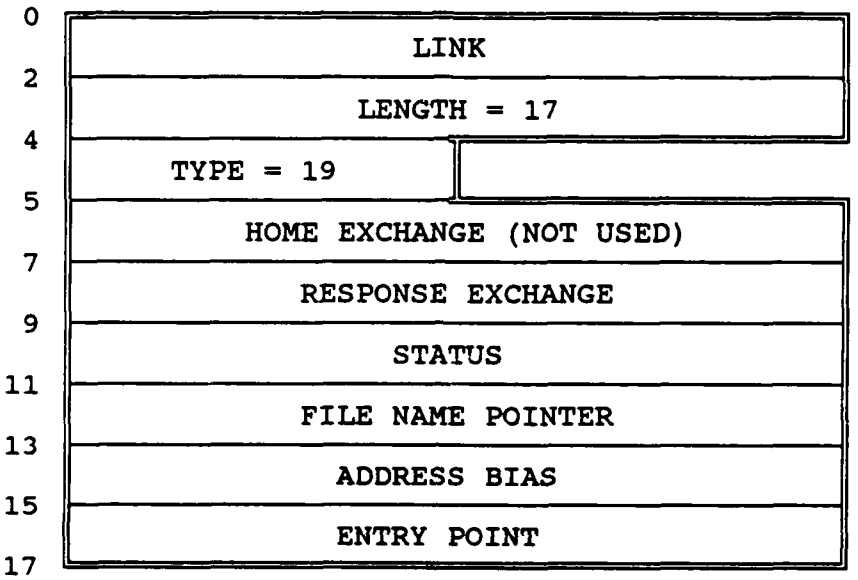
* The Loader accepts only absolute object code files with all EXTERNAL references satisfied. It is therefore impossible to load code inadvertently which is not suited for execution.

* The Loader can control the execution of other routines. Prior to the actual loading operation, an EXTERNAL routine R@LLOK is invoked (which is part of RXIROM in the RXISIS-II environment). This routine disables entry to the Monitor which is otherwise permitted to use the Loader buffer as scratch memory. Correspondingly, a routine R@LREL is called when the Loader is finished and its buffer can be used again by the Monitor.

* A recovery procedure for soft disk errors improves the reliability of the loading operations. In case of a disk read error, RQLOAD moves the head out one track, re-positions it to the correct track, and attempts to read again. If reading fails again, it steps the head one track [illegible] the center of the disk, back again, and tries again.

tain data. This cycle is repeated up to five times; only if the error persists, a disk error message is issued.

The interface between the Generic Loader and the surrounding system is compatible with the iRMX-80 LOAD task. The Loader is invoked by a request message sent to its entry exchange RQLDX; the message must have the following structure:



The calling task must set the LENGTH, TYPE and RESPONSE EXCHANGE fields, and, in addition, the FILE NAME POINTER and ADDRESS BIAS locations. FILE NAME POINTER must point to an iRMX-80 compatible 11-byte File Name Block; ADDRESS BIAS holds a value which is added to the loading start address specified by the disk file loaded (modulo 64K). RQLOAD returns two STATUS bytes and, in ENTRY POINT, a start address if the module loaded was a main module, or otherwise zero. The STATUS value returned corresponds to ISIS-II and iRMX-80 standards (compare Appendix 4).

The Loader task RQLOAD requires, in addition to the request message, parameters in two memory locations declared PUBLIC by RQLOAD:

RQLLBD Two-byte (ADDRESS) variable holding the Low Loader Boundary.

RQHLBD Two-byte (ADDRESS) variable holding the High Loader Boundary.

Loading is only permitted to addresses between and including the values stored in RQLLBD and RQHLBD. These boundaries may be changed at any time provided the Loader is not active while they are being changed. The default values set by the Loader's initialization sequence are 0001H for RQLLBD, and 0FFFFH for RQHLBD. It is therefore possible to load to the entire 64K address range from 0001H through 0FFFFH, with the exception of location 0000H (which is used for ROM anyhow and can therefore not reasonably be included in a loading operation).

The Loader uses a 256 byte (100H) buffer which must be kept in controller accessible RAM, and which may be used by other tasks or routines as scratch memory while the Loader is not active. (No permanently required data can be stored there.) In the CGCS controller computer, the Loader buffer is allocated in the top memory page, from address 0FFF0H through 0FFFFH.

Particularly for bootloaded systems, it is important to have as much memory as possible available for the code to be loaded. Still, some memory is required for the code and data structures of the task requesting the Loader's operations. It is possible, though, to locate at least data in an area which may be overwritten by the loaded code. This applies, for example, to the iRMX-80 File Name Block holding the name of the file to be loaded. This File Name Block can be kept anywhere in memory except in the lower 128 bytes of the Loader buffer. In general, all information which is no more needed once the new file was loaded can be kept in memory which may be overwritten.

3.3.6 Entry Points Into ROM Resident Code

In general, the RXISIS-II Initialization Code and the RXISIS-II Monitor (plus Confidence Test) are entered via vectors in the 8080 Restart area (compare Appendix 3). The corresponding addresses may be called or jumped to by user code, or the user code may issue the proper RST instruction. The following entry points are available:

RST 0	0000H	Monitor System Reset Entry Point.
RST 1	0008H	RXISIS-II Initialization and Re-Boot.
RST 2	0010H	Monitor Main (Breakpoint) Entry Point: Register contents are saved; PC is set to contents of Top of Stack.

RST 3 0018H Monitor Auxiliary Entry Point; see separate Table.

A set of parameters is required if the Monitor is entered via its Auxiliary Entry Point (RST 3). The value passed in register C (the first parameter of a call from PL/M) determines the Monitor function invoked, while the register pair D+E holds a parameter required for this function. (From PL/M, this parameter is passed as the second in a procedure call.)

The following switch parameters are permitted for C (all other values cause an error message):

C = 0 Submit Control to the Monitor: This entry point is used by the DEBUG switch under RXISIS-II. All processor registers, except the stackpointer and the program counter, are reset to zero. The stackpointer is set to the Monitor's own stack, and the program counter to the value passed in D+E.

C = 1 Error Entry Point: This entry point is invoked by RXISIS-II in the case of a fatal (disk) error. The contents of the register pair D+E are interpreted as an error code (corresponding to the ISIS-II and RMX-80 error codes, compare Appendix 4). An error message is generated with an extensive error identification text if the error code is a more common one; control is subsequently submitted to the Monitor via its Reset Entry sequence. The Exit ("E") command of the Monitor is therefore inoperative; it is the responsibility of a system which calls this entry point to close all open files prior to the Monitor call.

3.3.7 Configuration of the RXISIS-II System ROM

Due to the bank-switched operation of the system ROM, its configuration is not entirely straightforward. Essentially, the following steps are required for the preparation of the ROM resident software:

(1) The contents of both ROM banks have to be linked together separately. Bank 0 contains the Confidence Test and the Monitor, Bank 1, the iRMX-80 Nucleus, the Alternative Terminal Handler, the generic Loader, and the ROM resident part of RXISIS-II, RXIROM.

A special treatment is required for the preparation of the Confidence Test routines: The ROM checksum verification

sequence which is part of the Confidence Test in Bank 0 can obviously operate on the ROM bank only which contains the ROM test code, i.e., on Bank 0. In order to permit testing of ROM Bank 1, a duplicate of the ROM test routine is copied into RAM at execution time (after the RAM was tested) from where it can access ROM Bank 1. This duplicate routine must therefore be configured for an execution in RAM (the addresses between 3000H and 30FFH are reserved for this purpose) but moved into the ROM address range (from 0F00H through 0FFFH) during program configuration. This can be accomplished by means of an interactive Object File Editor utility (OBEDIT) which has been specially designed for the configuration of the RXISIS-II system ROM.

The iRMX-80 based routines in ROM Page 1 can be configured with the help of Intel's Interactive Configurator Utility ICU-80. ICU-80 generates an iRMX-80 Configuration Module and a Create Table (which hold information about the tasks and exchanges in the iRMX-80 system); the SUBMIT (batch) file created by ICU-80 requires some slight modifications, though, in order to accommodate the specific features of the RXISIS-II environment.

(2) The modules holding the contents of the two ROM banks have to be combined separately with code which is common for both banks. This applies to the bank switching sequences in the area of the 8085 Restart vectors (addresses 0000H through 003FH), and to the Monitor exit sequence which reactivates ROM Bank 1; this sequence is located at the high-address end of the ROM area (addresses 1FF8H through 1FFFH). Since ROM page switching is done by ROM resident code, it is essential that the page switching sequences are provided in corresponding addresses of either ROM bank in order to guarantee a correct continuation of the program code in the new ROM page after a bank switch.

(3) The contents of either ROM page are subsequently submitted to a checksum calculation; two two-byte checksums, each covering 4 KBytes of ROM, are deposited (by OBEDIT) at the addresses 0020H through 0023H.

(4) ROM page switching is effected by controlling the most significant address bit of the ROM chips with the SOD output of the CPU. Therefore, the first ROM chip holds the lowest 2 KBytes of Bank 0 in its lower half, and the lowest 2 KBytes of Bank 1 in its upper half, and so on. The two modules with the contents of ROM Banks 0 and 1, respectively, have therefore to be "sliced" into four sections of 2 KBytes each, and interleaved properly in order to result in the memory pattern shown in Fig. 8.

This procedure can be done again with the Object File Editor OBEDIT. The resulting disk file is, finally, programmed into the four 2732A EPROMs of the CGCS computer.

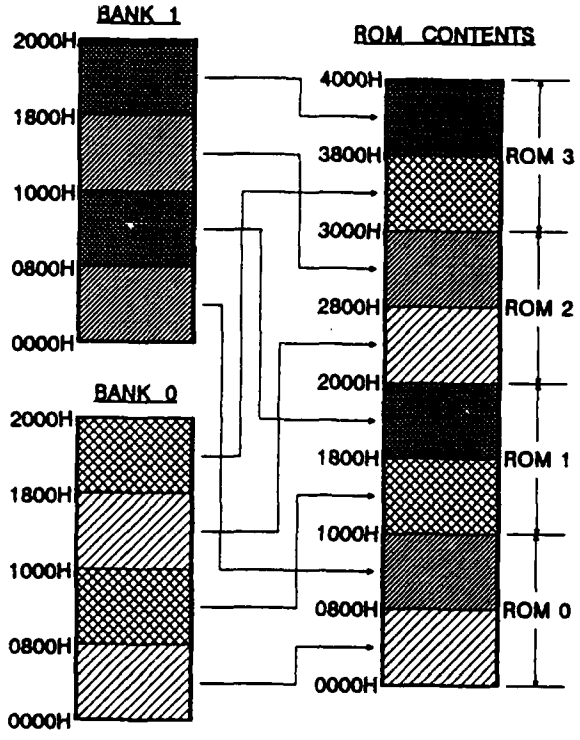


Fig. 8: Configuration of the RXISIS-II system ROM.

3.4 RXISIS-II

3.4.1 The Operation of RXISIS-II

RXISIS-II is an emulator for Intel's operating system ISIS-II (Intel System Implementation Supervisor) executed under Intel's iRMX-80 Real Time Multi-Tasking Executive for 8080/85 Processors on Intel's OEM Single Board Computer hardware. In general, RXISIS-II emulates the most important features of ISIS-II, and permits programs written for use under ISIS-II to run in a real-time environment. File notation and handling is identical for both systems, and files created under either system are compatible to the other. Some restrictions apply to the emulation of ISIS-II, though, since RXISIS-II is mainly intended as an auxiliary software package for genuine real-time process controllers where many development system oriented functions of ISIS-II are hardly required. This applies, in particular, to the possibility of batch processing with SUBMIT files. No features which support the re-routing of console input to a disk file have been implemented with RXISIS-II; it is, therefore, not possible to execute SUBMIT command files under RXISIS-II.

Additional restrictions apply to the size of memory available to programs executed under RXISIS-II (compare Fig. 7): While the top of user accessible memory is at 0F6BFH under ISIS-II, it had to be lowered to 0C7FFH in order to accommodate the RXISIS-II code; with the iRMX-80 Debugger loaded, MEMTOP is only at 92FFH. On the other hand, the program space available for programs under RXISIS-II starts already at 2800H rather than above the ISIS-II file buffers, which is typically at 3680H. In order to fully utilize the available RAM, specially relocated versions of some programs may therefore be used under RXISIS-II. Several reasons demanded that the above memory mapping was chosen for RXISIS-II:

(a) The disk I/O buffers internally used by RXISIS-II must be located in Multibus accessible RAM, i.e., at addresses above 4000H.

(b) RXISIS-II contains numerous iRMX-80 library routines which can hardly be subdivided in order to fill only certain memory locations, e.g., the area between 2800H and 3680H.

(c) RXISIS-II must provide not only the subroutine entry points of ISIS-II but also of the ISIS-II Monitor, which implies that addresses around 0F800H had to be reserved for entry points.

Since RXISIS-II is essentially considered a software tool, providing additional disk file handling capability for genuine real-time application systems, it is designed to be eventually overwritten and replaced by arbitrary real-time applications running under iRMX-80. Such application code is bootloaded from disk similar to RXISIS-II, and it may freely use the ROM resident routines of RXISIS-II. In general, the system always comes up under RXISIS-II; as far as the user is concerned, however, other real-time application systems can be invoked from RXISIS-II at any time just like ordinary ISIS-II programs.

Being based on the ROM environment described in the previous chapter, RXISIS-II is designed to make ample use not only of the ROM resident iRMX-80 routines but also of the Monitor. For high-level debugging purposes, the iRMX-80 Active Debugger may be loaded into the RXISIS-II system at any time; it becomes resident and can be activated and used as detailed in Intel's iRMX-80 User's Guide. Although the memory space available for utility and user routines is significantly restricted in this case due to the memory requirements of the Debugger, this feature adds powerful debugging aids.

In order to supplement the utility routines available under RXISIS-II, and in addition to ISIS-II BASIC, a special iRMX-80-based version of BASIC is available. This BASIC interpreter can be invoked from RXISIS-II but does not utilize the RAM resident RXISIS-II routines. Indeed, it is an entirely independent real-time system. This approach permits the most economic use of system memory and provides the iRMX-80-based BASIC with about 6.5 Kbytes more of free program and data storage space than its ISIS-II counterpart under RXISIS-II.

RXISIS-II is activated whenever a disk which contains its system files - RXISIS.BIN, RXISIS.CLI, and RXISIS.PSC - is installed in drive 0, and an "E" or "Q" command is executed from the Monitor, or when "Return" is pressed after the initialization sequence after a Reset. RXISIS-II is normally <u>not</u> reloaded when a program executed under its control terminates. RXISIS-II displays a sign-on message and loads its Command Line Interpreter from the file RXISIS.CLI. (The Command Line Interpreter shares memory with ISIS-II application programs, which makes it necessary to re-load it each time a program terminates.) RXISIS-II displays a hyphen ("-") as a prompt, similar to ISIS-II, and waits for a command to be entered. The command line editing and control codes listed in chapter 3.3.4.2 or in Appendix 5 can be used under RXISIS-II.

The execution of ISIS-II software under RXISIS-II is identical to ISIS-II. The reader may refer to Intel's "ISIS-II User's

Guide" to obtain full information. With only a few exceptions, RXISIS-II commands refer to the names of disk files which have to be loaded and executed. In contrast to ISIS-II, however, the names of these files are not identical to the commands to be entered by the operator; a file name extension is used to indicate to RXISIS-II the type of the program, and the way it is to be executed. Programs with the file name extension ".RXI" are qualified to run in a genuine real-time environment, which implies certain additional features which are not available under ISIS-II, such as type-ahead. The extension ".RXR" indicates that the program may be executed under RXISIS-II, but real-time operation is not permitted. In some cases, this is due to the insufficient stack size of programs which were only available as object code. Since interrupts may result in a stack overflow with these programs, all interrupts must be disabled while the actual program code is executed, and they may only be enabled during calls to RXISIS-II system routines. In addition, some programs like Intel's CRT-based full-screen editor CREDIT or the ISIS-II version of BASIC use input handlers of their own which are based upon single character console input rather than line-based input. It is essential that type-ahead is disabled for such programs to guarantee their proper operation. The ".RXI" and ".RXR" extensions are automatically appended to the user specified program name by the RXISIS-II Command Line Interpreter; neither of them need therefore be specified by the user.

Each command (with the exception of "@" and "DEBUG") is, indeed, interpreted by RXISIS-II as the name of a program file. If, for example, the user entered "MYPROG", RXISIS-II first searches for a file "MYPROG.RXI". If RXISIS-II finds such a file, it loads and executes it. Otherwise, RXISIS-II continues with a search for "MYPROG.RXR". A message indicating that type-ahead is disabled is output if "MYPROG.RXR" has been found and loaded. If this file does not exist either, RXISIS-II scans the disk directory for a file "MYPROG". Since a file without an ".RXI" or ".RXR" extension may be incompatible with RXISIS-II, great care must be taken in executing it; such programs are therefore not automatically executed after loading but control is vectored to the Monitor. The user may, in this case, choose whether he wants the program executed (with the Monitor's "G(o)" command), or whether he prefers to exit back to RXISIS-II with the Monitor's "E(xit)" or "Q(uit)" commands. (iRMX-80 and RXISIS-II system code and data in ROM are protected from being overwritten by incompatible program code.) Different files "MYPROG.RXI", "MYPROG.RXR", and "MYPROG" may exist in parallel on the same disk. By default, "MYPROG.RXI" will be executed. It is possible, however, to load and run also the other two programs if "MYPROG.RXR" and "MYPROG." are invoked, respectively. (The trailing period in the "MYPROG."

command would constitute a command error under ISIS-II. Under RXISIS-II, in contrast, it is used to explicitly access program files without a file name extension.) In both cases, control will be vectored to the Monitor after the respective program was loaded.

File and device names may be entered in upper- or lowercase characters.

3.4.1.1 Available Devices

The following device names are permitted under RXISIS-II:

:F0: ... Disk Drive #0: This drive must contain a valid RXISIS-II system disk, i.e., a disk containing the files RXISIS.BIN, RXISIS.CLI, and RXISIS.PSC. The device specification ":F0:" is assumed by default; it may be omitted.

:F1: ... Disk Drive #1

:CI: ... Console Terminal Input

:VI: ... Console Terminal Input; treated identically to :CI:

:CO: ... Console Terminal Output

:VO: ... Console Terminal Output; treated identically to :CO:

:LP: ... Line Printer

:TO: ... Line Printer; treated identically to :LP:

:BB: ... Byte Bucket: Pseudo output device for dummy output operations

3.4.1.2 Available Programs and Functions Under RXISIS-II

Programs which are not included in the following list are not necessarily RXISIS-II incompatible. The programs listed below have, however, been successfully tested under RXISIS-II.

Some of these programs (e.g., CREDIT or BASICI) use single character rather than line based input. Characters which are entered while these programs are not ready to accept input are added into the type-ahead buffer of the Terminal Handler which

may eventually overflow, causing an error beep. During some operation sequences, e.g., during data output in ISIS-II BASIC or while CREDIT refreshes a CRT screen page, the program operation is halted if any key on the console terminal is pressed, and resumed only after any key is pressed again. The program operation proper is, however, not affected by such effects.

3.4.1.2.1 Intel Supplied Utility and Development Software

The following Intel supplied utility and software development programs have been tested under RXISIS-II and were found to be fully compatible. For more information, please refer to Intel's "ISIS-II User's Guide" (unless otherwise specified).

ATTRIB File Attribute Display and Modification Program.

COPY File Copying Program.

CREDIT CRT Based Text Editor (see Intel's "ISIS-II CREDIT CRT Based Text Editor User's Guide"): a special CREDIT.MAC file is generally required which customizes CREDIT for the CRT terminal used. No type-ahead can be used with CREDIT.

DELETE File Deleting Program.

DIR Disk Directory Display Program.

LIB Library Manager Program.

LINK Program Module Linker Program.

LOCATE Program Module Locator Program.

RENAME Disk File Renaming Program.

3.4.1.2.2 Other Utility Software

The following additional utility routines which were originally written to extend the capabilities of Intellec Development Systems running under ISIS-II were adapted for RXISIS-II. Some of these programs were newly configured for RXISIS-II; the ".RXI" versions of these programs cannot be executed under ISIS-II since they use memory below the ISIS-II buffer area. For more information about the programs listed below, refer to Appendix 6 of this documentation, unless otherwise stated.

ADOC	Text Formatting Program (see Text Formatting Program "ADOC" Reference Manual, K. Riedling, February 1981).
ATTSET	File Attribute Modification Program.
CMPDSK	Disk Comparison Program.
COMP	Disk File Comparison Program.
COPYCP	Disk File Copying and Compare Program.
CPYDSK	Disk Copying Program.
CREATE	File Creation Program.
DIRFIL	Disk Directory Formatting Program.
DISOBJ	Object File Display Program (no documentation available).
HEXCHK	Hexadecimal File Dump Program.
LIST	File Listing Generation Program.
SHOW	File Display Program.

3.4.1.2.3 Programming Languages Under RXISIS-II

Although arbitrary program source files may be generated or edited under RXISIS-II (with CREDIT), the use of compilers is, unfortunately, prohibited by the restricted memory size. It is possible, though, to execute Intel's 8080/85 Macro Assembler ASM80 (and, in turn, to link and locate program modules with the LINK and LOCATE programs). ASM80 could, in fact, be executed under RXISIS-II with full real-time capabilities (i.e., with the ".RXI" extension); still, the Cross Reference generation overlay ASXREF has insufficient stack resources, which requires that ASM80 be executed in the Restricted Mode (i.e., without type-ahead) if the XREF switch is used.

There are two versions of Intel BASIC interpreters available for use under RXISIS-II: The ISIS-II BASIC Interpreter can be invoked with "BASICI"; it offers all functions known from ISIS-II but a relatively limited workspace area (11848 bytes). An iRMX-80 based BASIC can be loaded with "BASIC"; its workspace is larger (18268 bytes) but it does not support some functions of the ISIS-II BASIC (e.g., program line editing).

iRMX-80 BASIC is, indeed, not executed under RXISIS-II but constitutes a self-contained complete iRMX-80 system which can be loaded from RXISIS-II but replaces RXISIS-II. It can only be left via the Monitor (or via a system reset), in contrast to ISIS-II BASIC which can be exited via the standard "EXIT" command. BASIC programs generated with either version are fully compatible.

ASM80 8080/85 Macro Assembler (see Intel's "ISIS-II 8080/85 Macro Assembler Operator's Manual").

BASIC iRMX-80 based BASIC (see Intel's "iRMX-80 BASIC Reference Manual").

BASICI ISIS-II based BASIC (see Intel's "BASIC-80 Reference Manual").

3.4.1.2.4 Special RXISIS-II Functions and Programs

The following functions and programs are used exclusively by RXISIS-II; they cannot be executed under ISIS-II (although there are some similar ISIS-II functions).

@: Disk File Display Utility.

This disk file display utility is executed by the RXISIS-II Command Line Interpreter rather than by an explicitly loaded program. It is invoked by entering

@ [<device name>]<file name>

Only disk files can be displayed with "@". The first page of the specified file is presented immediately. The operator can continue the file display line by line by pressing the space bar on the console terminal; pressing any other key triggers the display of a new page. The display procedure can be exited by pressing the Escape key. Tab characters are resolved properly. Lines whose lengths exceed the screen width (80 columns) are subdivided; the subdivision is indicated by an arrow ("--->") to the left of the continuation line. Non-printable characters are replaced by question marks ("?"). Since all console input is interpreted as control signals for the display utility, no type-ahead input can be entered during the execution of this function.

DEBUG: Program Execution Under Monitor Control.

DEBUG is, in fact, not a program call but sets an internal switch in RXISIS-II. Its use is similar to the DEBUG function of ISIS-II; further information can be obtained from Intel's "ISIS-II User's Guide".

FORMAT: Disk formatting utility.

FORMAT permits to format blank disks. The program is invoked with the command

FORMAT [<device name>]<label>

where <device name> may be either :F0: or :F1:, and <label> any sequence of one to six alphanumeric characters which may be followed by an optional extension of a period and one to three alphanumerics. <device name> determines the drive on which the disk is to be formatted, and <label>, the future disk label. <device name> may be omitted; in this case, the disk in drive 0 is formatted. The program confirms the data given with the call, and requests operator actions where necessary. Note that only the ISIS-II but not the RXISIS-II system files are created with the FORMAT call. An error message pertaining to missing RXISIS-II files is therefore issued after the execution of FORMAT if the disk in drive 0 was formatted. FORMAT re-boots RXISIS-II after its execution. Any type-ahead is therefore lost.

RMXDBG: iRMX-80 Debugger.

The program RMXDBG loads and activates the iRMX-80 Active Debugger (compare Intel's "iRMX-80 User's Guide"). The top of user accessible memory is moved down to 92FFH. RMXDBG makes itself resident in memory and returns control to RXISIS-II; arbitrary RXISIS-II compatible programs which do not exceed the available memory space may be loaded and executed. The Debugger remains active until RXISIS-II is re-booted.

@	File Display Utility.
FORMAT	Disk Formatting Utility.
RMXDBG	iRMX-80 Debugger.

3.4.1.3 Executing Programs Under RXISIS-II

The first routine which is automatically invoked upon power-up is the built-in Confidence Test, or, more precisely, the memory test sequence of this routine (compare chapter 3.3.2). The Confidence Test vectors control to the System Restart sequence of the Monitor which issues a proper sign-on message. Any hardware system reset happening at a later stage directs control immediately to the Monitor, bypassing the Confidence Test. Memory contents are thus preserved from destruction by the memory test routine.

An RXISIS-II system disk, i.e., a disk containing the files "RXISIS.BIN", "RXISIS.CLI", and "RXISIS.PSC", should now be inserted into drive #0. Pressing the "RETURN" key immediately after the "System Restart" message was output makes the system load RXISIS-II from disk. Control is vectored to the Monitor if any other key is pressed (compare chapter 3.3.1). A hyphen ("-") appears as an operator input prompt when RXISIS-II is ready for command input.

At this stage, any program compatible with RXISIS-II may be invoked. Since RXISIS-II obtains console input via the Alternative Terminal Handler, all line editing and control features described for the Alternative Terminal Handler (compare chapter 3.3.4.2) apply fully.

There are two basic operation modes for the execution of user programs, namely, Regular and Debug Mode.

(1) Regular Mode: Upon entry of the device name (if applicable; only ":F0:" and ":F1:" are permitted) and, without intervening spaces, the file name, the program file is loaded, and control is submitted to the loaded code unless an error was detected by RXISIS-II during program loading. It lies in the responsibility of the loaded program to return control to RXISIS-II upon completion.

(2) Debug Mode: Debug Mode is entered if the above command is preceded by the switch "DEBUG" (and an intervening space). In this case, the specified file is loaded, but control is not transferred to the loaded code. Instead, the Monitor

is invoked. The user is now at liberty to display or modify code or data prior to executing the program (compare chapter 3.3.1). If the loaded code was a main module, its start address is already available in the Monitor's Program Counter location; in this case, the program can be started with a simple "G" (GO) command without additional parameters. Execution breakpoints may be set if necessary. The user must take care to return control to RXISIS-II after program execution (compare chapter 3.3.1.4). A "DEBUG" command entered without a subsequent file name permits to access the Monitor without loading a program.

Three general types of programs may be invoked under RXISIS-II. The simplest type is represented by most of the utility programs listed in chapters 3.4.1.2.1 and 3.4.1.2.2. These programs are, from the point of view of the resident iRMX-80 system, a simple quasi-subroutine extension of RXISIS-II. Such programs may be executed in arbitrary order, without many precautions since they belong, from iRMX-80's point of view, to the task RXIROM which executes RXISIS-II.

Some programs, however, require one or more additional tasks to be created, e.g., "FORMAT" and "RMXDBG". These additional tasks are usually built by the part of the disk loaded code which is an extension of RXIROM. Such programs must by no means be simply overwritten by new program code or data even if they are no more required since the additional tasks they contain have been included into the iRMX-80 task scheduling. The easiest way to get rid of such no more required tasks is to re-create the entire iRMX-80 system by re-booting RXISIS-II.

Some applications, finally, do not need the ISIS-II interface of RXISIS-II altogether; they are more efficiently configured without RXISIS-II. Such programs (e.g., iRMX-80 BASIC or genuine real-time application systems like the CGCS) are kept in an overlay which is loaded instead of RXISIS-II rather than in addition to it after an iRMX-80 system restart. RXISIS-II was configured to permit the loading of such entire systems as if they were simple utility programs like those described in chapter 3.4.1.2; aside from a more prolonged disk activity, the operator will hardly notice any difference. Such systems are activated by loading a dummy program under RXISIS-II (e.g., "CZOCHR.RXI") whose only purpose is to vector control to a dedicated part of the Command Line Interpreter code. This routine replaces the extension ".RXI" by ".BIN", stores the modified file name for the bootloader, and calls the bootloader. In our example, the file "CZOCHR.BIN" is therefore loaded instead of "RXISIS.BIN" after iRMX-80 was restarted.

3.4.2 The Programming Interface of RXISIS-II

3.4.2.1 Preparation of RXISIS-II Programs Without Additional Tasks

The system routines provided by RXISIS-II behave - with some minor differences - identically to the corresponding ISIS-II and ISIS-II Monitor routines. The appropriate documentation in Intel's "ISIS-II User's Guide" applies therefore; the library modules of the ISIS-II SYSTEM.LIB may be used to access RXISIS-II functions.

No special care is required for the preparation of programs intended for running under RXISIS-II which do not introduce new tasks, and the instructions given in Intel's "ISIS-II User's Guide" can be followed exactly. Programs which are to be executed under RXISIS-II must not use memory above 0C7FFH (or, above 92FFH if they should run with the iRMX-80 Debugger installed); on the other hand, they may access memory from 2800H upwards. This boundary is not affected by the number of disk files used by the program. Note that RXISIS-II sets the stackpointer to the current top of memory before it submits control to the user program loaded; code or data located next to MEMTOP may therefore be overwritten if the user code uses the stack without redefining the stackpointer. (On the other hand, programs which do not redefine the stack may return to RXISIS-II with a simple RETURN machine instruction.)

In order to maintain compatibility between ISIS-II and RXISIS-II, the program code start address should, if possible, be chosen at 3680H according to the ISIS-II rules. The stacksize must be at least 24 bytes plus the stack required by the program itself in order to permit a non-restricted execution mode. Programs which should be executed under RXISIS-II must be configured as main programs (otherwise, the Monitor is invoked after the routine was loaded, and the start address has to be entered manually).

No special procedures are required for the linkage and locating of such programs; all PUBLICs required to satisfy the EXTERNAL references are contained in the standard ISIS-II library SYSTEM.LIB which should therefore be linked in as the last library. (The library RXISIS.LIB must be included after SYSTEM.LIB if either of the new routines SETMTP or EXICHK is used.)

The following ISIS-II and ISIS-II Monitor routines and functions are implemented under RXISIS-II:

ISIS-II:	OPEN	
	CLOSE	
	DELETE	
	READ	
	WRITE	
	SEEK	
	LOAD	
	RENAME	
	EXIT	
	ATTRIB	
	RESCAN	
	ERROR	
	WHOCON	
	SPATH	
MONITOR:	CI	(Console input)
	CO	(Console output)
	LO	(Printer output)
	CSTS	(Check console status)
	IOCHK	(Check system i/o configuration)
	IOSET	(Set system i/o configuration)
	MEMCK	(Return top of user accessible memory)

The following restrictions and differences to ISIS-II apply:

(1) No line-edited disk files are permitted.

(2) RESCAN can therefore be applied to the console input only.

(3) Only the console CRT terminal is permitted as a console device. This prohibits the use of SUBMIT files.

(4) It is not possible to re-define the console device either with CONSOL or with IOSET. The IOSET routine provided is only a dummy routine without any effect.

Two routines are available in RXISIS-II in addition to the ISIS-II system routines, namely, SETMTP and EXICHK. Their entry points (and the entry points of all above RXISIS-II routines) are kept in the library RXISIS.LIB.

SETMTP: This routine permits to change the top of the user accessible RAM. The address specified with the call is the highest location which can be overwritten during program loading. The routine requires one address type parameter (the MEMTOP value) if called from PL/M, or the MEMTOP value in the B-C register pair if called from assembly language.

EXICHK: This routine returns the System Exit flag which is set by the Monitor to 0FFH upon an "E" command (compare chapter 3.3.1.4) and which is otherwise reset to zero. The flag is a byte parameter returned in the A register; the processor's zero flag is set accordingly. EXICHK may be called by routines which perform lengthy operations without invoking any of the (RX)ISIS-II functions. An ISIS-II EXIT call should be performed if EXICHK is found set.

3.4.2.2 Preparation of RXISIS-II Programs With Additional Tasks

Evidently, compatibility to ISIS-II need no more be maintained for such programs. Additional tasks can be created dynamically by the appropriate RQCTSK calls issued by the program's main routine (which continues the RXISIS-II task RXIROM). Such systems must be linked with the PUBLICs of the ROM resident system which are contained in the library RXIPUB.LIB. The program should verify whether the version of RXIROM resident in the system when it is called is the same version with which it was linked. (The version code of RXIROM is the value of the PUBLIC parameter RXIVER.) In some cases, it may be necessary to link such programs also to the PUBLICs of the disk resident part of RXISIS-II (e.g., to prevent the directory-based Disk File System routines from being linked in a second time). An un-purged version of RXISIS.BIN is, for such purposes, available under the name RXISIS. The program should, however, also check the RXISIS-II overlay version number in this case which is the value of the PUBLIC parameter RXIVSN. (The RXIVER and RXIVSN values of the current system configuration are stored in RAM at the locations 0FEAEH and 0FEAFH (for RXIVER), and 0FEACH and 0FEADH (for RXIVSN), respectively (see Appendix 3).

After their execution, programs containing additional tasks must no more return to RXISIS-II unless they made sure by lowering MEMTOP accordingly (via a SETMTP call) that the code of the task(s) created by them is located in memory which is protected from being overwritten, or unless all tasks and exchanges created by the program are deleted. Otherwise, such programs should be terminated with an RST 1 instruction which restarts RMX-80 and re-boots RXISIS-II, rather than with an ISIS-II EXIT call.

3.4.2.3 The Preparation of Real-Time Application Systems

The rules given in the preceding chapter apply also to real-time application systems which include RXISIS-II. Systems which do not need the ISIS-II emulation (like the CGCS) should rather be configured as independent iRMX-80 based disk overlays. Such overlays are loaded in two steps:

First, a dummy program with the name of the system and the extension ".RXI" is loaded which vectors control to a dedicated routine of the RXISIS-II Command Line Interpreter. The Command Line Interpreter replaces the ".RXI" file name extension by ".BIN" and passes the resulting program name to the ROM resident initialization/loader code. Such a dummy program is available on the RXISIS-II Software Development Disk under the name "BOTLOD.OBJ" (or, e.g., as "CZOCHR.RXI"); it need only be renamed appropriately.

Second, the system proper is loaded by the ROM resident bootloader. The bootloader checks the ROM system version under which the overlay was created and issues a fatal error message if this version differs from the current one. The bootloader expects the two-byte ROM version code (which is supplied by RXIROM as the value of the PUBLIC variable RXIVER) in the locations 0FEAEH and 0FEAFH, where it must have been deposited by the system overlay loaded.

Application system code may be loaded into RAM between and including 2800H and 0FEADH. Memory below 2800H is reserved for the ROM resident tasks; the memory locations from 0FEB0H through 0FEEFH contain the Disk Controller's buffer, and RAM from 0FFF0H through 0FFFFH is used by the Loader. The latter locations may be overwritten by application systems if the Loader is no more needed; still, they are also partly used by the Monitor. Data stored there may therefore be mutilated when the Monitor is invoked. (Compare Appendix 3.)

In general, real-time application systems can be configured according to the rules for bootloaded iRMX-80 systems. The ROM resident part of the task RXIROM loads the disk overlays, checks for loader errors and for the correct linkage version, and starts executing the loaded code beginning with its entry point address (the overlay must be a main module). A special routine, RMXOVL, has been provided in RXIROM.LIB which must be linked in as the first module when the application system overlay is created, instead of the START module of the iRMX-80 Loaded System Library LOD8xx.LIB. The commission of RMXOVL is to create the tasks and exchanges contained in the Create Table of the overlay, and to delete the task RXIROM which is supposed to be no more required by the application system.

Furthermore, RMXOVL is configured to provide the ROM system version code in 0FEAEH and 0FEAFH. (Application systems may use a similar entry routine which does not delete RXIROM, and continue this task with an arbitrary function. This has been done, for example, in the CGCS where RXIROM continues as the Command Interpreter task.)

The LINK call for configuring the overlay should therefore contain the following items:

```
An iRMX-80 Configuration Module
RXIROM.LIB (RMXOVL) (or an alternative module)
Application specific modules
LOD8xx.LIB (The Bootloader Library for the iSBC used)
All iRMX-80 extensions required
RXIPUB.LIB
RMX8xx.LIB
BOTUNR.LIB
UNRSLV.LIB
PLM80.LIB
```

The Configuration Module and a configuration SUBMIT file may be created with Intel's Interactive Configurator Utility ICU-80. Still, the ICU-80 created SUBMIT file must be edited to comply with the above structure.

Disks exclusively used for a real-time application system need not necessarily contain the files RXISIS.BIN and RXISIS.CLI. Still, they must provide the Cursor Positioning overlay RXISIS.PSC if they are to be mounted on drive 0. Due to the loading approach chosen, application system overlays may be located on and invoked from either drive.

3.4.2.4 Use of ROM Resident Routines by Application Systems

Aside from the Terminal Handler and Loader tasks, the RXISIS-II ROM contains several routines which are used by RXISIS-II but may also be called by application system tasks:

RXCFNB: Create File Name Block.

This routine parses an input buffer for a valid file name whose start address is submitted as a parameter. It returns a completion code which specifies the device type or possible error conditions, and a File Name Block which can be directly used by the iRMX-80 Disk File System routines. If called from an assembly language routine, RXCFNB also provides a pointer to

the first character after the file name. Leading spaces in the input buffer are ignored by RXCFNB. Strings following a colon (":") are interpreted as device names; only the device names listed in chapter 3.4.1.1 are permitted. Strings without a leading colon are supposed to be names of disk files on drive 0 (":F0:" is appended internally). Invalid device names (all entries different from the names listed in chapter 3.4.1.1), invalid file names consisting of more than 6 alphanumeric characters, illegal extensions longer than 3 alphanumeric characters, and missing extensions (file name terminated with a period but without an extension) are reported. The File Name Block is undefined or partly undefined in this case. It is also undefined if a non-disk device was specified. A special code is returned if the string "DEBUG" (without extension) was detected. A file name is considered terminated if the first not alphanumeric character (not necessarily a blank) is detected; the register pair B+C is returned as a pointer to the location of this character.

CALL FROM PLM:

```
status = RXCFNB (.buffer,.file$name$block)
```

PARAMETERS FOR ASSEMBLY LANGUAGE CALLS:

```
    A ... Completion Status Byte (O)
    B+C . Input String Start Address (I)
          Address of First Character After String (O)
    D+E . File Name Block Start Address (I)
```

The following status codes are returned:

```
00H ... Device = :CO: or :VO:
01H ... Device = :CI: or :VI:
02H ... Device = :LP: or :TO:
0FH ... Device = :BB:
10H ... Device = Disk; File Name With Extension.
11H ... Device = Disk; File Name Without Extension.
20H ... Device = Disk; File Name "DEBUG".
40H ... Illegal Device Name.
80H ... Illegal File Name.
81H ... Missing File Name.
82H ... Illegal Extension.
83H ... Missing Extension.
```

Valid data are contained in the File Name block if the codes 10H, 11H, or 20H are returned.

RXCEXT: Create Extension

Similar to RXCFNB, RXCEXT parses an input buffer for a file name extension string (one to three characters). The parameters and completion codes of RXCFNB apply analogously.

RXCNEX: Create Null Extension

In contrast to the above routines, RXCNEX does not add data to a file name block specified with the call but replaces a possible extension within the file name block by binary zeros (corresponding to no extension). For reasons of compatibility, the same parameters and completion codes apply as for RXCFNB; although no input string is actually required a dummy such parameter must be specified with a call from PL/M.

RXCRMX: Create RMX-80 Overlay System

This routine replaces the START module of the Loaded System Library LODxxx where xxx corresponds to the type of the processor board used (80-24, 80-30). It uses the Create Table RQCRTB specified for the overlay system; such a table may be created by Intel's Interactive Configurator Utility ICU-80. The address of this Create Table must be passed as a parameter to the RXCRMX call. (Since RQCRTB is declared PUBLIC by ICU-80, a "Multiplely Defined Publics" error message referring to it is issued during the system linkage; this error message can be ignored.) RXCRMX first creates all exchanges and subsequently all tasks specified in the Create Table.

CALL FROM PLM:

```
CALL RXCRMX (.RQCRTB)
```

PARAMETERS FOR ASSEMBLY LANGUAGE CALLS:

B+C . Address of the Create Table RQCRTB

3.4.2.5 Other Utility Routines in the Library RXIROM.LIB

The library RXIROM.LIB holds, in addition to the routines already discussed above, a number of auxiliary routines which may be called by application programs. Since these routines are not included in the system ROM, they have to be linked in explicitly when the user program is configured.

RMXOVL: Start Module for iRMX-80 System Overlays.

This routine has already been mentioned and discussed in chapter 3.4.2.3.

LNKDBG: Link RMX-80 System Overlay with Debugger.

This module has to replace RMXOVL if the iRMX-80 Active Debugger is to be included in an iRMX-80 real-time application system. It includes the Active Debugger's Static Task Descriptor and some initialization sequences required for the execution of the Debugger (particularly, the generation of a RAM resident vector to the breakpoint entry point of the Debugger). The module RST5VC in RXIROM.LIB must also be included with LNKDBG. The particular structure of RXISIS-II prohibits that the modules constituting the iRMX-80 Debugger can be directly requested if the Configuration Module is created with ICU-80. The corresponding libraries must be specified with the ICU-80 "LINK" command.

CRTFNB: Create File Name Block.
CRTEXT: Create File Name Extension.
CRTNEX: Create Null Extension.

These routines are, in fact, the predecessors of RXCFNB, RXCEXT, and RXCNEX, respectively (see chapter 3.4.2.4). They differ from the ROM resident routines insofar as they perform a more stringent check of the file name string. With CRTFNB, a file name must be terminated with a blank (space or control character), while with RXCFNB any non-alphanumeric character is accepted as the file name terminator. (Non-alphanumeric characters within the six places of a file name, or the three of an extension, cause an error code of 80H and 82H, respectively, if the file name string is parsed with CRTFNB.) The same applies to CRTEXT. The programming interfaces are identical to those of the corresponding "RXC..." routines.

4. The Operation of the Czochralski Growth Control System

4.1 Basic Operation Concepts of the CGCS

4.1.1 General System Design

The Czochralski Growth Control System (CGCS) was designed as an interactive real-time process control program whose initial purpose was to replace the conventional analog controller supplied by Cambridge Instruments for the CI-358 LEC puller. From its initial commission of emulating the analog system, the digital controller was further enhanced by the addition of advanced deterministic and, finally, heuristic control features. The key characteristics of the CGCS can therefore be grouped according to the level and quality of control.

(1) Emulation of the analog controller:

The CGCS is connected to the puller's output signals in parallel to the analog system, permitting both controllers to monitor a growth run in parallel (compare Fig. 2). The following CGCS functions can be considered a replacement of the analog controller:

* Display of measured data and setpoints: All relevant measured data (as listed in chapter 2.4.1) are displayed permanently on the console terminal; Fig. 9 shows a (simulated) display screen.

```
08-10-87  21:19:54  Run ID: Demonstration Screen  MACRO   System Time: 27:16:22
-------------------------------------------------------------------------------
                 Actual:   Setpoints:  |  Mode: Automatic       Length:    85.45
Diameter (D):    83.73     82.00  82.00|
                                       |  Ramping:    2/20      Condit.:    1/8
Temp. 1 (T1):    23.65     23.63  23.50|
Temp. 2 (T2):    23.98     23.95  23.80|  Weight:    2348.     Diff.Wt.:  1.476
Temp. 3 (T3):    23.39     23.36  23.25|  Seed Pos.: 246.7     Cruc.Pos.: 23.89
                                       |  Base Temp: 20.19     Gas Press: 297.6
Power Limit (PL):          80.00  80.00 ----------------------------------------
                                       |                  Actual:   Setpoints:
Seed Lift (SL): 9.003      9.000  9.000|  Seed Rot. (SR): 4.997     5.000  5.000
Cruc Lift (CL): 1.487      1.492  1.500|  Cruc Rot. (CR): -30.0     -30.0  -30.0
-------------------------------------------------------------------------------
Power In/Out:   47.37/45.29       49.12/48.28       45.40/42.12      Contact: *32*
-------------------------------------------------------------------------------
2881H=    -28       28C9H=    -31       2842H= 0.001250   36F9H= 23.67148
set prop10 -20 300
macro
***** Executing Macro MACRO  *****
deb c rcrset 4
Please Command:
comm This is a demonstration screen with arbitrarily invented data_
```

Fig. 9: Console screen of the CGCS.

The console screen shown in Fig. 9 displays, in general, the following items:

(1) Date, time, and run identification in the top line. Two times are displayed, namely, the actual time (in 24 hours format), and an internal system time which starts at zero when the system is initialized, and can count up to 95 hours, 59 minutes, and 59 seconds. (It wraps around to zero after 96 hours, and starts counting up again.) The top line holds, in addition, a space between the run identification and the system time where the name of a Macro Command (see below and chapter 4.5) will be displayed while it is executed.

(2) A regularly updated display of measured system parameters and of setpoints. Two columns are provided for the display of the setpoints: The left column holds the currently valid setpoints, whereas the right column displays the final setpoints which may differ from the current setpoints if a parameter is being modified by the system. If a parameter is set by the output of a controller (e.g., the heater temperature in diameter controlled mode), the right column indicates a bias value input to the controller. Parameters which can be entered as setpoints are, in addition to their full names, identified in the screen display by the two character abbreviation with which they are to be identified. The system was designed to accommodate a three-zone heater. Therefore, three heater temperatures and three pairs of power values are displayed. (In the current implementation at ASU, only one heater channel is meaningful; the measured data for the second and the third channel have been tied to those of the first channel.) There are two output power values for each heater channel, referred to as "In" and "Out"; the "In" values specify the percentage of maximum power which is input to the power controller, while "Out" gives the actual output power; both are scaled to lie between 0 and 100. (The "In" values are, in fact, calculated and output by the CGCS, whereas the "Out" data are measured data input by the CGCS. "In" and "Out" refer to the power controller, not to the CGCS.)

(3) Internal system status information: This information comprises the number of parameters being "ramped" (i.e., being modified linearly between their initial and intended final values within an arbitrary

time), and the number of Conditional Macro commands pending, against their respective maximum values (20 and 8, respectively). Furthermore, the operation mode (see MODE command) is displayed close to these two values in the top right corner of the screen.

(4) Command echoes and system messages: While the upper part of the CGCS output screen is in a fixed format and updated in a random access mode, the echo and message area (five lines in the bottom third of the display) is scrolled up as information is added in the bottom line. The echoes of operator entries are displayed there, and messages issued by the system are directed there, too. In addition, the same area is used by some commands for the display of menus or auxiliary information. The scrolled portion shrinks to four lines if auxiliary data display is requested with the DEBUG Continuously command. In this case, the top line of the scrolled portion is used for the DEBUG output.

(5) Command prompt line: All operator actions are requested in the last line but one on the screen.

(6) Input area: The bottom line is reserved for the currently entered command line. In general, the RXISIS-II rules apply to the entry and to the editing of commands.

* Data logging and recording: All parameters displayed on the console screen can optionally be recorded on disk for later analysis. One record (which contains the values of more than 50 parameters) can be written to disk at intervals ranging from one second to 255 seconds. The entire dialogue between the operator and the CGCS can be copied to a line printer or to a disk file, with the time appended at which each line was generated.

* Control of the "primary" system parameters: The primary parameters - heater temperature(s), motor speeds, and a power limit for the heater(s) - can be modified interactively in either an absolute mode (i.e., by specifying a target value), or in a relative mode (i.e., by entering the intended positive or negative change). In either case, the system can be instructed to adjust the parameter instantaneously to its final value, or to "ramp" it gradually between its initial and intended final values within an arbitrary interval. Digitally implemented PID control loops are used for maintaining the controlled parameters at their respective setpoints.

* Diameter control: Based on the standard weighing method, the diameter of the crystal grown is determined with an algorithm which takes into account the buoyancy of the part of the crystal which is immersed in the boric oxide encapsulant. Depending on the operation mode of the CGCS chosen, the heater temperature can be controlled to maintain the crystal diameter at its setpoint. (Further details of the controller operation are given in chapter 4.1.2.)

(2) Advanced features of interactive control:

The following features of the CGCS facilitate and enhance control over the puller; although they increase the degree of automation significantly over the one offered by the analog Cambridge Instruments controller, they are not sufficient yet for an entirely automated operation.

* "Variables": Any arbitrary system parameter can be identified by a symbolic name, and displayed and modified exactly like primary parameters. (A directory of Variable names is kept in a disk file; its size is only limited by the available disk space and by the time it takes to locate a given Variable.) This feature permits access to intermediate results and to controller parameters which may be modified dynamically according to the requirements of the process.

* "Macro" commands: All commands actually pertaining to the crystal growth process can optionally be recorded in a disk file, with the time (relative to the start of the recording) appended at which each command was issued. The resulting command file may be edited (or created) with a Macro Command Editor program which runs under RXISIS-II, and used as an input for subsequent growth runs. Commands recorded in the Macro command file are executed exactly with the sequence and timing of the original run. Since Macro commands can be referred to by a single-word command, and since they can be started arbitrarily, they constitute a powerful feature of combining complex sequences of commands, thus relieving the operator from more complex command entries, improving the reproducibility of the process, and avoiding operator errors. Commands originating from a Macro command file may be interspersed with commands entered on the console; the resulting stream of commands may be recorded on disk again, which constitutes kind of a learning ability of the system. Macro commands may even invoke other Macro commands, which permits to concate-

nate a series of Macro commands for a more complex operation. (The current structure of the CGCS does, however, not permit subroutine-type Macro calls. A Macro command which invokes another Macro command is therefore preempted.)

The command flow in the CGCS is schematically depicted in Fig. 10; commands entered on the console and from a Macro command file are pre-processed by the Command Interpreter and Macro Command Input tasks, respectively, and submitted to the Command Executor task which constitutes the interface to the actual process control tasks. All commands sent to the Command Executor are advanced in chronological order to a Command Output task which writes them to disk if instructed to do so.

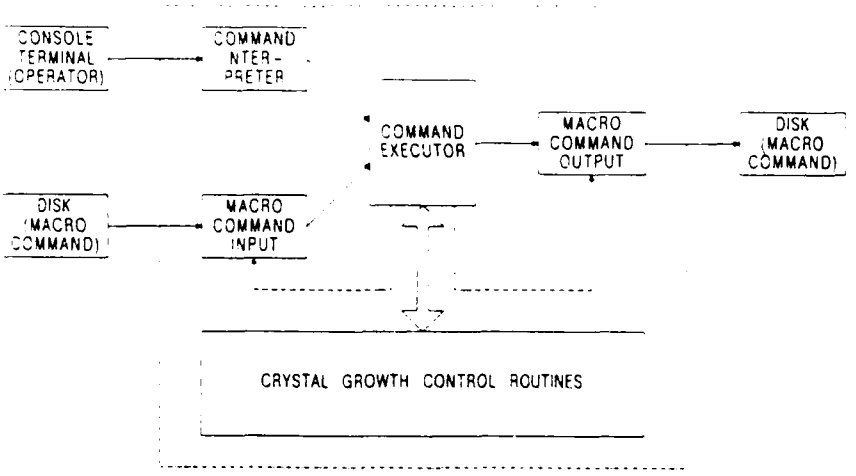


Fig. 10: Command execution in the CGCS.

(3) "Intelligent" control:

The above features are strictly deterministic, similar to most known crystal growth automation approaches. They are not sufficient, though, for a fully autonomous growth control since they are not flexible enough to allow for the fluctuations which are typical for crystal growth processes. While some operator interaction is therefore required for initiating operations or sequences of operations in other "automated" crystal growth controllers, the CGCS endeavors to replace the decisions made by the operator by internally generated decisions, which requires features akin to artificial intelligence.

"Intelligent" control is effected in the CGCS by the conditional execution of Macro command files, i.e., by the execution of a Macro command if and when a Variable assumes a specified relation to a given constant (e.g., greater than or equal). The CGCS maintains a table of such Conditional Macro commands which may be issued at any time by the operator, or by another Macro command; the conditions for the execution of each of these commands are checked periodically until either the condition is found met, and the Macro command is executed, or until the Conditional Macro is removed from the table by a pertinent command. Since each Macro command may issue one or more Conditional Macro commands, the CGCS can be programmed to handle properly even relatively complex operations like seeding in an entirely autonomous way.

In contrast to the above "vertical" hierarchy within the CGCS, we can distinguish five "horizontal" operation modes each of which comprises a higher degree of closed-loop control. The operation modes are identified by a numeric parameter and a keyword name as follows:

0 - Monitoring: No control at all is performed by the CGCS in this mode.

1 - Manual: Closed-loop control comprises the four motor speeds, and the temperatures of up to three heaters. The CGCS does not effect closed-loop diameter control.

2 - Diameter: In addition to the motor and heater closed-loop control, the CGCS controls the crystal diameter via the heater temperature (compare chapter 4.1.2). The plain differential weight, without anomaly correction, is used for diameter evaluation.

3 - Diameter/ASC: "ASC" stands for "Anomaly Shape Control": A correction similar to the approach used in the analog Cambridge Instruments Anomaly Shape Control board is used for pre-processing the differential weight prior to the diameter calculation (compare chapter 5.3.2.2.2). Aside from this correction step, mode 3 is equivalent to mode 2.

4 - Automatic: Automatic mode comprises all features of Diameter/ASC mode, plus closed-loop control of the crucible lift speed which maintains the growth interface at a constant location within the heater's hot zone (compare chapter 4.1.2).

4.1.2 Control Loops in the CGCS

The fundamental operations of the CGCS, namely, the control of the speeds of the four motors for seed and crucible lift and rotation, and of the power supplied to the heater or heaters, utilize conventional closed-loop control methods which are generally based on PID controllers realized with a generic PID routine. This routine is invoked with dedicated parameters for each control loop. In addition to standard proportional, integral, and derivative control, the generic PID controller features several modes of output limiting and "windup" protection (which enhances its dynamic response if the controller incurs a limit condition); the possibility to add a bias value to the output of the PID controller allows for feed-forward operations, and for small corrections of setpoints which are basically determined by other sources.

The standard control loop for each of the four motors is outlined in Fig. 11: The primary control of the motors is done by the analog circuitry which came with the Cambridge Instruments controller. Under digital control, the setpoint for these analog motor controllers is supplied by the D/A converter outputs of the CGCS, rather than from a potentiometer on the analog console. Basically for the compensation of nonlinearities and offset errors of the analog motor controllers, digital PID loops are used to pre-process the signals finally submitted to the analog system in order to make the actual speeds exactly match their corresponding setpoints. A combined feed-forward and PI control approach can be used to optimize the performance of the entire control loop. (Using an analog hardware-based rather than a digital software-based technique for the primary motor control guarantees a sufficiently smooth and fast operation without overburdening the digital system.)

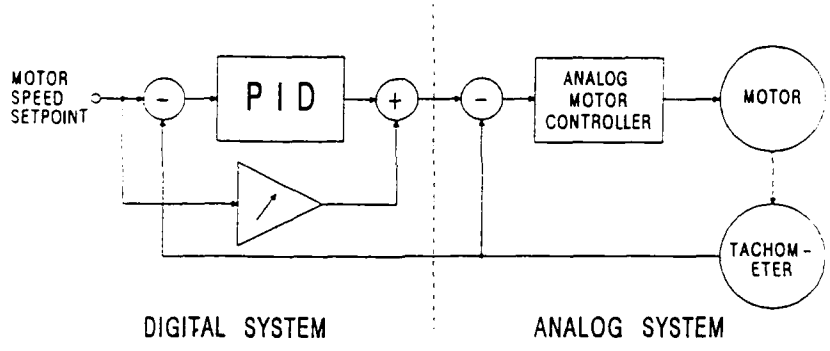


Fig. 11: Control loop for one of the four motors in the CGCS (analog/digital and digital/analog conversions are not explicitly shown).

With regard to the diameter control method generally applied to the growth of compound semiconductors, temperature and diameter control are closely related to one another (Fig. 12): In "manual" mode, i.e., without closed-loop diameter control, a temperature setpoint value is compared to the digitized output of the thermocouple which monitors the temperature of the heater; the resulting difference is submitted to a PID controller whose output controls the power setpoint of the analog heater SCR controller. In "automatic" closed-loop diameter controlled mode, the heater temperature setpoint is modified by the output of a superimposed diameter control loop. In contrast to the standard Cambridge Instruments diameter controller which controls the heater temperature according to the deviation of the first derivative of the crystal weight ("differential weight") from a given setpoint, the CGCS first calculates the actual crystal diameter, and uses it as an input to the diameter controller. This permits a more straightforward and understandable operation of the controller. Since the CGCS was designed for up to three heater zones, three independent temperature and diameter controllers according to Fig. 12 have been provided in the controller program.

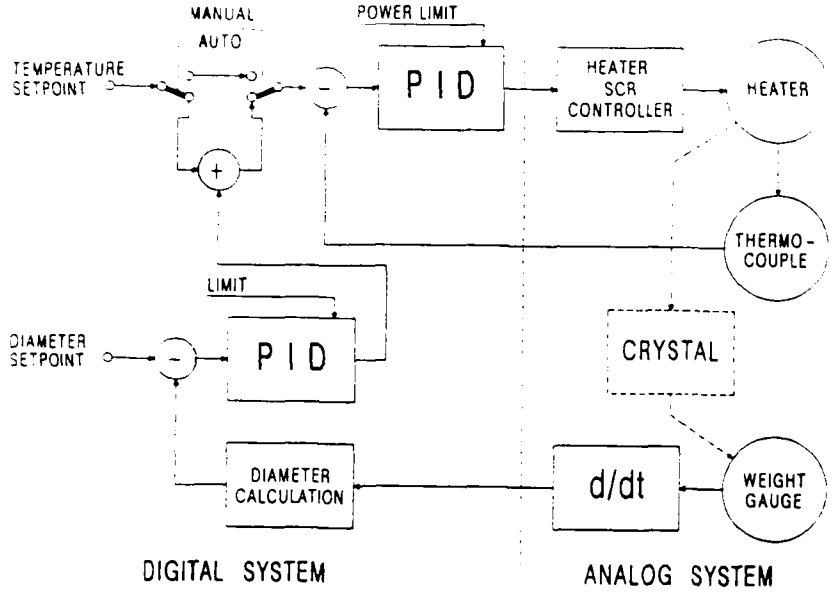


Fig. 12: Heater temperature and crystal diameter control loops (analog/digital and digital/analog conversions are not explicitly shown).

An auxiliary control loop (Fig. 13) can optionally be applied to the vertical speed of the crucible: Since the level of the semiconductor melt in the crucible drops during a growth run according to the amount of material solidified, the interface between the solid crystal and the melt would change its position within the heater, which is liable to cause growth instabilities, unless the crucible is lifted exactly by the amount of the melt drop. On conventional pullers, the crucible lift speed is set to a fixed value which is calculated under the assumption of an ideally cylindrical crystal with constant diameter. The CGCS, in contrast, computes a setpoint value for the crucible position as a by-product of the diameter evaluation routines, essentially by determining the amount of melt already used up by the crystal; this setpoint is compared to the actual crucible position, and the resulting error signal is used as an input for a PID controller whose output is superimposed on the crucible speed setpoint.

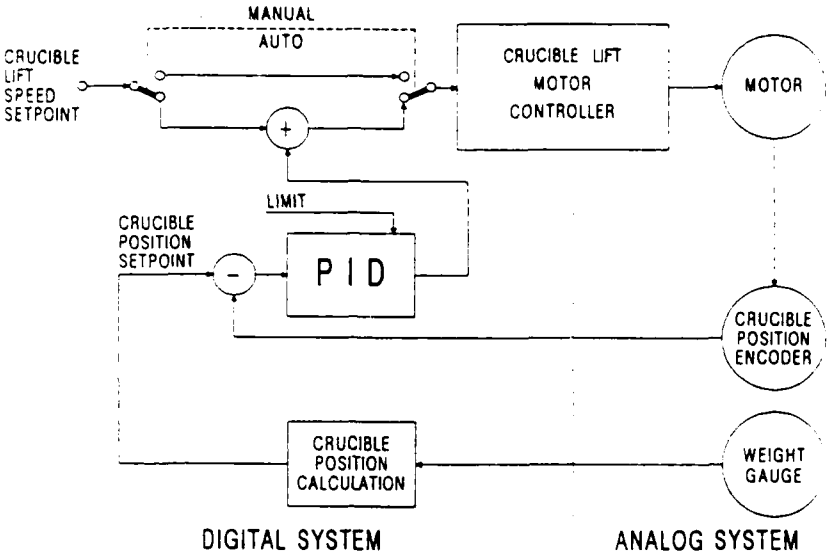


Fig. 13: Crucible position control loop (analog/digital and digital/analog conversions are not explicitly shown).

4.1.3 Diameter Evaluation in the CGCS

The actual diameter of the crystal, and a number of auxiliary parameters like the crucible position setpoint, the growth rate, and the crystal length grown, are calculated by the CGCS once every ten seconds. In addition to the differential weight, several other measured parameters are used as inputs for these computations, as shown schematically in Fig. 14.

The diameter evaluation approach used in the CGCS is based upon the differential weight signal supplied by an analog differentiator circuit. After its A/D conversion, this signal is submitted to digital low-pass filtering; an anomaly compensation analogous to the approach used in the Cambridge Instruments Anomaly Shape Control board may be applied to it. The current diameter of the crystal is calculated from this differential weight using the actual growth rate (i.e., the difference between the seed and crucible lift speeds plus the speed with which the semiconductor melt drops when it is consumed by the crystallization process). A full compensation

for the buoyancy in the boric oxide encapsulant is provided; the diameter evaluation routine keeps track of the shape of the part of the crystal next to the solidification interface (to be accurate, of the last 75 millimeters of the crystal), and calculates the volume immersed in the encapsulant and the height of the boric oxide layer from this information. This approach permits the use of actual physical parameters of the system (like densities, dimensions, and speeds) rather than the modified parameters required in conventional analog growth.

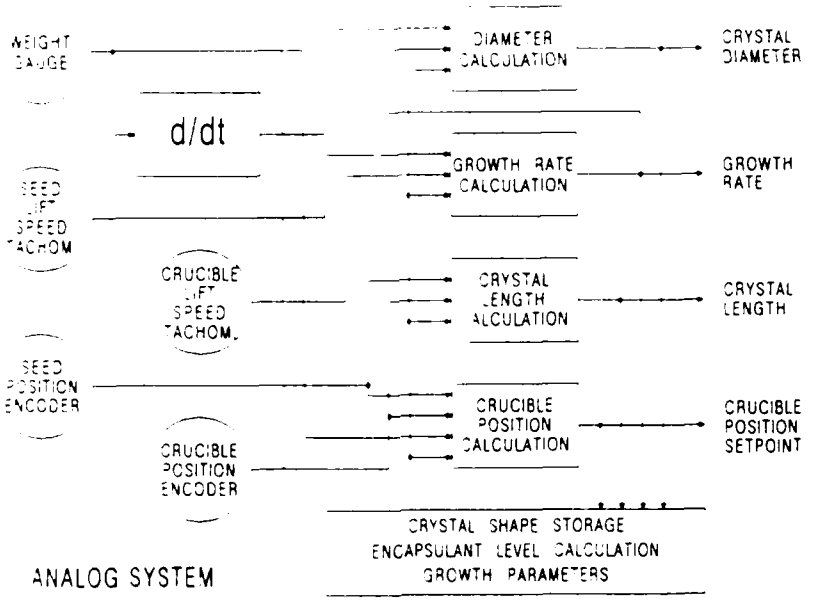


Fig. 14: Block diagram of the evaluation algorithms for the crystal diameter, the growth rate, the crystal length grown, and the crucible position setpoint (analog/digital and digital/analog conversions are not explicitly shown).

The diameter and crucible position evaluation algorithms which are used throughout the major part of a crystal growth run are based on the following assumptions:

(1) The crucible is a straight right cylinder.

(2) The amount of boric oxide encapsulant remains constant.

(3) The semiconductor melt fills the entire diameter of the crucible, and material added to the crystal reduces the height of the semiconductor melt in accordance with the conservation of the total mass (melt plus crystal).

While assumption (2) is reasonably justified in the case of gallium arsenide throughout the entire growth process because the boric oxide encapsulant does not wet the crystal, this does no more apply to the other two assumptions towards the end of a growth cycle: The transition between the crucible wall and bottom is always a bevel with a finite radius; and the semiconductor melt tends to contract itself due to surface tension and recedes towards the center of the crucible if its amount drops below a certain limit. In an extreme case, the above assumptions have to be amended as follows:

(1) The semiconductor melt forms a cylindrical disk with constant thickness whose diameter (rather than thickness) decreases in order to supply the material being solidified in the crystal.

(2) The gap which opens up therefore between the semiconductor melt and the crucible wall is filled with boric oxide encapsulant, which reduces the effective boric oxide height as the crystal grows.

The diameter evaluation algorithms used in the CGCS are capable of handling both extreme cases, and any arbitrary intermediate stage, according to the value of the Variable ALPHA. An ALPHA value of 1 corresponds to the first set of conditions (when the semiconductor melt fills the entire crucible diameter), whereas a value of 0 conforms with the second set (i.e., extreme melt recession). Values for ALPHA between 0 and 1 permit to model an intermediate stage between the two extremes in a heuristic mode: Most likely, the disk formed by the receding melt <u>does</u> reduce its thickness when the melt is used up by the growing crystal; the speed with which it does so may, however, be considerably less than during the regular growth. An ALPHA value less than 1 but still greater than zero will therefore be appropriate during the final growth stages. Since crystal growth will always start under conditions corresponding to the first set of assumptions, ALPHA is initialized with 1, and remains at this value unless it is explicitly set to a different value.

The RESET command is closely linked to (and required by) the diameter evaluation routines. It initializes the shape information required for the buoyancy compensation under the assumption of a cylindrical seed with the diameter specified with the INITIALIZATION command (or sequence) which passes through the entire boric oxide encapsulant layer, and it provides initialization values for the crystal length and weight calculation. Furthermore, a RESET command resets ALPHA to 1 and cancels all effects of a possibly different previous ALPHA value.

4.2 Starting the CGCS

The Czochralski Growth Control System (CGCS) is started from RXISIS-II, but it is a genuine real-time process control program which is independent of the RXISIS-II environment.

NOTE: The system needs the CGCS system disk permanently in drive 0. The operator must by no means exchange this disk unless prompted to do so (see the EXCHANGE command). To be save in the (improbable) case of a disk error on the system disk, a second system disk should be kept at hand which must, however, be of the same system version. The system will crash inevitably at the attempt to install a disk in drive 0 which holds a different CGCS version!

The CGCS is invoked from RXISIS-II like any other RXISIS-II function, namely, via a call by its name, CZOCHR. Provided the disk in drive 0 holds a valid copy of the CGCS, it will be loaded, and a sign-on message is displayed. During this initialization, the system checks whether the A/D Converter hardware is installed and operational, and it enters into a Test mode if this is not the case. A message "Test Run" is displayed if the A/D converter does not respond properly, and input from the A/D converter and output to the D/A board are suppressed. This feature permits testing of the CGCS software in an environment which does not provide the hardware interface to the puller; running the CGCS with disabled inputs allows, in addition, to simulate input parameters for testing purposes. (Analog I/O can also be suppressed under software control in a fully equipped hardware environment; compare chapter 4.7.2.)

Among may other initialization chores, the CGCS disables the BREAK key on the console terminal, and enforces a duplication of Monitor output on the printer. All inadvertent entries into the Monitor program will therefore show up in a Documentation printout.

During the entire growth run, the CGCS checks the integrity of its program code periodically. RXISIS-II should be re-booted, and the CGCS re-started as soon as possible if a memory error is reported in order to avoid unforeseeable reactions of the system.

Subsequently, the CGCS prompts for the current date (which is not updated even if a run extends beyond midnight) and time, and for an arbitrary run identification code. Date and time must be entered in the format displayed by the CGCS; the seconds can, however, be omitted (they will be assumed to be zero

in this case). The date and time entries must be acknowledged by the operator; a plain "Return" in response to the confirmation prompt will accept the data displayed in the top line of the screen.

During the initialization of the system, some commands are automatically performed by the CGCS, thus saving the operator typing and making sure that all required information is entered. The system permits to open a Documentation output file (otherwise done with the DOCUMENTATION command), and requests a set of constants (see the INITIALIZATION command). Finally, the Command Interpreter's prompt "Please command:" is displayed, and the CGCS enters its regular operation mode.

4.3 Command Set of the CGCS

4.3.1 General Remarks

The operation of the CGCS is determined by independent commands which are interpreted by the Command Interpreter (one of the CGCS's iRMX-80 tasks). There are two types of commands, namely, Internal, and Macro commands. Internal commands are directly executed by the program; they provide the basic control functions. Macro commands, in contrast, are in fact disk files which are read when their name was entered as a command. These disk files hold, in turn, one or more Internal commands, with a time information attached. These Internal commands are therefore not only executed in the order in which they were recorded on the file but also with the same timing. Macro command files can be generated either by directly recording the commands entered on the console during a growth run, or with the Macro Command Editor COMMED.

Internal commands are generally invoked interactively, i.e., the operator is prompted for more information if necessary. Some of the Internal commands can be entered in one single command line, which simplifies the dialogue between the system and the operator significantly. All items which may be specified together with the command keyword are listed in the summary of Internal commands in chapter 4.3.2. Commands which are likely to affect either the growth process proper, or essential functions of the CGCS, require, in general, a reconfirmation of the data entered by the operator with an explicit acknowledgement response (e.g., "Y(es)"); any other entry, including "Return" only, cancels the command.

All valid Internal commands and the descriptions of their purposes are listed below in alphabetical order. It is obviously not possible to give a similar list of Macro commands since they may be freely defined by the operator. It is therefore up to the operator to keep a record of his Macro commands and of their functions.

The following syntax is used for the Internal commands:

CAPITALS constitute the part(s) of the command which must be entered exactly as specified.

lowercase parts of the command keyword are optional. They are specified here for clarity and may also be typed in but are ignored by the Command Interpreter.

Items in angular brackets < > have to be replaced with the ap-

propriate contents, e.g., a parameter value or a Macro command name.

Items in square brackets [ ] are optional and may be omitted.

Items included in braces { } and separated by a vertical bar | are optional but one item of the list must be specified.

Items must be separated by at least one space (except within file names).

Note: Commands may be issued in arbitrary order. A command is, however, only recognized when the prompt "Please command:" is displayed!

4.3.2 Summary of Internal Commands

```
CALCulate [{R|I|H}]
CHANge [{D|Tn|SL|CL|SR|CR|PL|<varname>}[<value> [<time>]]]
CLEAr [<varname>]
COMMENT [<arbitrary text>]
DATA
DEBug [C [<varname> [{1|2|3|4}]]]
DEBug [C [<hexaddr> [{A|I1|I2|R|H1|H2|H4} [{1|2|3|4}]]]]
DEBug [D [{<varname>|<hexaddr>}]]
DEBug [M [<varname>]]
DEBug [M [<hexaddr> [{A|I1|I2|R|H1|H2|H4}]]]
DEBug [O [{1|2|3|4}]]
DEBug [R [{<varname>|<hexaddr>}]]
DEBug [S [{<varname>|<hexaddr>}]]
DIRectory [{0|1}]
DISPlay [<varname>]
DOCUmentation
DUMP
END
EXCHange [{0|1}]
EXIT
FILEs
HELP or ?
IF [<varname> [{<|=|>}[{<|=|>}] [<value> [<macro>]]]]
INITialize
MODE
PLOT [{<varname>|<hexaddr>} [{1|2|3|4|5|6|7|8}]]
QUIT
RESEt [<initial weight> <initial length>]
RESTore
SET [{D|Tn|SL|CL|SR|CR|PL|<varname>} [<value> [<time>]]]
STARt
```

4.3.3 Comprehensive Description of the Internal Commands

CALCULATE: This command permits to calculate the sum, the difference, the product, and the quotient of two numbers. The input format and the treatment of the numbers depends on a switch entered with the command: The switch is "R" for floating-point ("REAL") numbers, "I" for integers (which must lie between -32768 and 32767), and "H" for hexadecimal values (e.g., memory addresses) which have the same numeric range as integers. Input values are explicitly requested in any case. The result is displayed in decimal and hexadecimal form, with the internally used hexadecimal format for floating-point numbers if applicable.

CHANGE: This command permits to modify the value of one of the nine primary system setpoints (crystal diameter, three heater temperatures, seed and crucible lift and rotation speeds, and power limit), or of an arbitrary system Variable (see chapter 4.7 and Appendix 11). CHANGE determines the current value of the specified parameter and adds the input value to it, thus permitting relative changes. Since the actual execution of the command is kept separate from the operator interface, the actual value of the target parameter may differ from the one displayed during the processing of the command if the target parameter is being ramped when the command is issued. Setpoints which are used as an input to a controller (e.g., the Temperature setpoints in Diameter controlled modes), are displayed with the values output by the controller. CHANGE permits a smooth transition of the parameter between its current and its final values by allowing a transition time during which the parameter is ramped (see remarks about parameter ramping in chapter 4.4). The transition time may range from zero to 9999 minutes (in fact, longer transition times are possible but cannot be displayed any more). The shortest non-zero transition time is one second; this value is used for all non-zero transition time values less than one second (0.017 minutes). The CHANGE command may be completely entered in one line, or in any combination of items. It may be recorded to and executed from a Macro command file.

CLEAR: The command CLEAR removes pending Conditional Macro commands from the Conditional Command queue. It may be used to branch between Macro commands if a condition specified with an IF command is not met within a given time. There are two types of CLEAR commands: An unconditional

CLEAR which removes all pending Conditional Macro commands, and a Selective CLEAR which cancels only those Conditional commands which refer to the Variable specified with the CLEAR command. CLEAR can be recorded to and executed from Macro command files.

COMMENT: This command inserts one line of comment into the Data output file. The comment line is tagged with the operation mode, time, and length grown information and embedded between the (binary) records in the Data file, thus permitting the correlation between arbitrary events and the data recorded. Even if no Data file is in use, the comment line is recorded in the Documentation output. (In fact, the COMMENT command is the only one to provide arbitrary text in the Documentation output.)

DATA: The DATA command permits to open or close the Data output file. It offers the operator to open a Data file if there is no open such file, and it permits to close the Data file if it is invoked while a Data file is open. After a disk error, the file which was involved in the error is flagged as "inactive". Not reactivating an inactive file is equivalent to closing it. The functions of DATA may be also accessed through the FILES command.

DEBUG Continuously: One member of the DEBUG command group, the DEBUG Continuously command permits the continuous display of the values of up to four system Variables. The data output provided is updated at the same rate as the fixed screen output (once every five to six seconds). (In fact, the memory locations specified with DEBUG Continuously are sampled once every second; their values are also recorded in the Data file.) Data can be selected for display either by specifying a Variable name, or by submitting the hexadecimal address of the memory location(s) whose contents are to be displayed. In the latter case, an additional format information is required since DEBUG does not know what kind of data resides at an arbitrary storage location in memory. The display formats available are ASCII (A), interpreting one byte at the specified address as a (printable) character, one and two byte decimal integers (I1 and I2, respectively), decimal floating-point (REAL - R), and one, two, and four byte hexadecimal representation (H1, H2, H4). Finally, one of the four DEBUG output channels (numbered 1 to 4) must be specified to which the output is to be directed. (Channels 1 to 4 are displayed in the DEBUG output line on the console from

left to right.) The DEBUG Continuously command may be completely entered in one line, or in any combination of items. It may be recorded to and executed from a Macro command file.

DEBUG Display: The DEBUG Display command displays the contents of one or several adjacent memory locations which have been specified either by a Variable name, or by a hexadecimal address. (For displaying the contents of a Variable in its standard representation, the DISPLAY command is probably more convenient.) The four bytes starting at the given address (or part of them) are displayed as ASCII characters, in hexadecimal notation, as one and two byte decimal integers, and as (four byte) floating-point numbers. The command may be completely entered in one line, or in any combination of items.

DEBUG Modify: This command permits to modify one to four bytes in memory whose starting address must be specified either with a Variable name, or as a hexadecimal number. The program knows how many bytes have to be modified to change the value of a Variable specified by name, but the data format has to be submitted separately if a hexadecimal address is used. The formats available are ASCII (A), interpreting one byte at the specified address as a (printable) character, one and two byte decimal integers (I1 and I2, respectively), decimal floating-point (REAL - R), and one, two, and four byte hexadecimal representation (H1, H2, and H4). The program displays the current contents of the specified location(s), and prompts explicitly for a new input value. With the exception of the new value, the entire command or parts of it can be entered in one command line. (For changing Variables specified by name, the SET and CHANGE commands are probably more convenient; in addition, they offer the ramping feature which is not supported by DEBUG.) The DEBUG Modify command can be recorded to and executed from a Macro command file.

DEBUG Off: While DEBUG Continuously turns on the output of Debug data, DEBUG Off turns it off again. The location (1 to 4) which is to be turned off must be specified. The command may be entered in one or in two lines. It may be recorded to and executed from a Macro command file.

DEBUG Resume: This command affects the internal operation of the system. It should only be used for debugging purposes. Therefore, no further information is given here.

DEBUG Suspend: This command affects the internal operation of the system. It should only be used for debugging purposes. Inconsiderate use of this command may disable the CGCS entirely. Therefore, no further information is given here.

DIRECTORY: The DIRECTORY command displays the contents of the directory of the specified disk. In addition to the file names, the disk label and the numbers of sectors in use and free on the disk are displayed. Note: The actual number of sectors in use may be much greater if a file is open for output on the specified disk. The actual number of used sectors cannot be determined, though, since it is an internal parameter of the operating system. The numbers displayed for the used and free sectors are, however, preceded by a ">" and a "<" sign, respectively, in this case. The command may be entered in one line.

DISPLAY: This function displays the value of a Variable submitted as a parameter with the call. The command may be entered in one line.

DOCUMENTATION: A call to DOCUMENTATION permits to switch on or off the Documentation output on the printer or on a disk file. DOCUMENTATION offers to open a Print file if no such file is open, and to close it if it is open. During the file opening procedure, DOCUMENTATION permits to set the interval between Data Dumps to the Documentation output (compare command DUMP). Any arbitrary interval between 1 and 255 minutes may be specified; periodic Data Dumps may be disabled altogether. After a disk error, the file which was involved in the error is flagged as "inactive". Not reactivating an inactive file is equivalent to closing it. The DOCUMENTATION routine is automatically invoked when the system is started; it may also be accessed from the FILES command.

DUMP: This command initiates a dump of 21 system parameters (essentially, of the measured data) to the Documentation output. In addition, it triggers one record written to the Data file.

END: The END command is the official way to terminate a command record in the Control Output file (which eventually may be used as a Macro command file). Although no more entries are added to the Control Output file after an END command, the file remains open, and the next record may be started at any time with a START command. (This permits to use one Control Output file throughout a growth run to which certain command sequences are recorded; the records in it can be separated into several Macro command files using the Macro Command Editor. Note, however, that an END command preempts a Macro command file used for input regardless of whether there are more commands after the END command or not.)

EXCHANGE: This command permits to exchange a defective or full disk safely. It closes the files on the specified disk which are still open, prompts the operator to substitute a new disk, and re-opens all files on the new disk which were open on the old disk when the operator indicated to the system that the new disk was installed. Since the output files are opened with the same names on the new disk, any file with an identical name on the new disk is overwritten. In addition, the output files need some editing because control structures used on the Data and Control Output files are not provided by EXCHANGE. (It is sufficient to concatenate the two output files with the ISIS-II/RXISIS-II COPY command, or to concatenate the second part of a Data or Macro command file with a separately generated file header.) Note that a Macro command will be preempted which is being read from a disk which is to be EXCHANGEd.

EXIT: The only regular way to leave the CGCS is the EXIT command. Depending on the current operation mode, the EXIT command "cleans up" the controller. It stops the lift motors if the puller is under the control of the CGCS, reduces the heater power to zero within six hours (unless the power is already zero), stops the rotations, and relinquishes, finally, control to the analog controller. Several safety procedures prevent the accidental execution of this function.

FILES: This command displays the current status of the Print, Data, and Control Output files and their names if the files are open. Subsequently, it permits to open or close one of the three files, entering the respective DOCUMENTATION, DATA, and Control Output file handling routines.

After a disk error, the file which was involved in the error is flagged as "inactive". Not reactivating an inactive file is equivalent to closing it.

HELP: The HELP command (or, alternatively, a simple question mark ("?")) provides a set of command menus on the screen. The menus displayed comprise a summary of the Internal commands, the currently available Macro commands, and an extensive explanation of each command. The Macro command list and/or the extensive help display may be skipped if not needed.

IF: This command permits the conditional execution of a Macro command (it does not work with Internal commands). The Macro command specified with the IF call is executed if and when a condition is met which is based on the numeric relation between a Variable and a constant which are submitted as parameters of the IF call. The numeric relations may be "greater than" (">"), "equal to" ("="), "less than" ("<"), or any combination of two of these three ("<>" stands for "not equal"). The order of the relation characters does not matter; "=>" is identical to ">=" and means "greater than or equal to". Eight (8) Conditional Macro commands may be pending at a time; any Conditional command issued while the maximum number of commands are pending is ignored, and a pertinent error message is displayed. The command may be completely entered in one line, or in any combination of items. It may be recorded to and executed from a Macro command file.

INITIALIZE: This command permits to assign values to certain system parameters which cannot be (easily) changed otherwise since they are kept in memory in a pre-processed form to facilitate control operations. The values set with INITIALIZE are the diameters of the crucible and the seed, the amount of boric oxide used, and the densities of the solid crystal, the crystal melt, and the boric oxide melt. Since these values are, in most cases, hardware dependent constants anyhow, INITIALIZE offers default values which can be accepted with a plain "Return", or overwritten by new data. INITIALIZE is automatically executed when the system is started; it must be called during a growth run when the crystal is melted back partly, and growth is resumed with a full-diameter crystal within the boric oxide melt. In this case, the diameter of the crystal must be specified as a seed diameter, in order to provide

a correct diameter evaluation after a subsequent RESET call.

MODE: The MODE command permits to select one of five operation modes which are numbered 0 through 4. Each mode is a inclusive set of the functions of the preceding one. Mode 0 ("Monitoring") provides monitoring without control, Mode 1 ("Manual"), a basic control but no closed-loop diameter control. The latter is possible with Mode 2 ("Diameter") which, however, does not include an anomaly compensation. Mode 3 ("Diameter/ASC") provides anomaly compensation, and Mode 4 ("Automatic"), in addition, a Crucible Lift control which is based on the exact amount of melt withdrawn from the crucible during the crystal growth. Each mode change is reported by the system, and an automatic Data Dump is triggered. The MODE command may be recorded to and executed from a Macro command file.

PLOT: The PLOT command permits to output continuously (similar to the DEBUG Continuously command) the values of up to eight locations in memory which can be specified by Variable names or by absolute hexadecimal addresses. While DEBUG Continuously routes its output to the operator console and the Data file, the PLOT output is directed to eight spare channels of the D/A converter which are connected to a suitable chart recorder. PLOT can only handle Variables which are in INTEGER*2 notation, which applies to all measured parameters and control output signals, and to a number of internal system parameters (compare chapter 4.6 and Appendix 11). A number of auxiliary locations were provided which hold "expanded" values of parameters of which only a narrow numeric range is of interest. For further information on the PLOT command, refer to chapter 4.6. The PLOT command may be recorded to and executed from a Macro command file.

QUIT: The QUIT command permits to preempt a currently active Macro command.

RESET: The proper operation of the diameter evaluation routines requires a RESET command at the beginning of the actual growth. The RESET command resets the length grown counter and the weight output to zero or to values specified with the call, and initializes the internal data structures of the diameter routines. It is indispensable to issue such a command after each INITIALIZE command

(including the one automatically performed at the beginning of the CGCS operations), and after each irrecoverable "Speed overflow" error, when the puller is again in a well-controlled condition and growth can resume. (Otherwise, no new diameter output is generated, and diameter control is not possible.) A RESET command which sets the crystal length and weight to zero is automatically generated if necessary when the operation mode is changed to one of the diameter controlled ones (Mode 2 through 4). It is possible to maintain the current length and weight values with a RESET command either by answering the pertinent questions accordingly if in the interactive mode, or by specifying a value for the parameter to be maintained which is less than twice its most negative value (i.e., less than -16000 for the crystal weight, and less than -1200 for the crystal length). The RESET command may be recorded to and executed from a Macro command file.

RESTORE: The RESTORE command restores the console output if it was corrugated, which can happen very easily if one of the function keys on the console terminal is pressed inadvertently, or if the "Return" key is pressed while the cursor is in the bottom line of the screen, e.g., after the entry of a full input line of 80 characters. It does not affect the actual control operations of the CGCS.

SET: This command permits to modify the value of one of the nine primary system setpoints (crystal diameter, three heater temperatures, seed and crucible lift and rotation speeds, and power limit), or of an arbitrary system Variable (see chapter 4.7 and Appendix 11). It sets the specified parameter to the input value, thus permitting absolute changes. SET permits a smooth transition of the parameter between its current and final values by allowing a transition time during which the parameter is ramped (see remarks about parameter ramping in chapter 4.4. The transition time may range from zero to 9999 minutes (in fact, longer transition times are possible but cannot be displayed any more). The shortest non-zero transition time is one second; this value is used for all non-zero transition time values less than one second (0.017 minutes). The command may be completely entered in one line, or in any combination of items. It may be recorded to and executed from a Macro command file.

START: This commands starts the recording of commands in the Control Output file. If no such file is open, START permits to specify and open a Control Output file. Command times recorded in the output file are relative to the time of the START command. (For example, a SET command issued 35 seconds after the START command will be executed 35 seconds after the Control Output file was invoked as a Macro command during a later run.)

4.4 Parameter Ramping

Parameters entered with the SET and CHANGE commands may be ramped linearly between their current values and the final values specified with SET or CHANGE. Arbitrary ramping times between 1 second and 9999 minutes may be used. Up to 20 parameters (primary system setpoints or arbitrary Variables) may be ramped at a time, no matter whether the pertinent commands were entered from the console, or from a Macro command file. The number of parameters which are ramped at a given time is displayed on the console screen. Note: A SET or CHANGE command requesting parameter ramping which is issued when already 20 parameters are being ramped will be executed instantaneously, without ramping. Watch therefore the number of ramped parameters carefully when you use extended ramping and/or Macro commands. A SET or CHANGE command referring to a parameter which is already being ramped does not increase the number of ramped commands. Parameter ramping can be halted by commanding CHANGE <parameter> 0 0 (change the parameter by 0 within 0 minutes).

4.5 Macro Commands

All operator entries input when the "Please command:" prompt is displayed are first compared to the list of the Internal commands. If no match is found between the first four characters of the operator input and any one of the Internal command names, the CGCS assumes that a Macro command was requested, and searches the system disk in drive 0 for a file with an extension ".CMD" whose name matches the operator entry. Therefore, the following rules apply to Macro command names:

(1) Macro command names may consist of one to six alphanumeric characters; the first character must be alphabetic.

(2) The first four characters of the Macro command (three characters if the command begins with "DEB") must not match any internal command name. (Note, though, that commands whose keywords are shorter than four characters have their names padded to the right with spaces. The name "SETPNT" is therefore a perfectly legal Macro name.) Macro names which are part of a Conditional command are excepted from these restrictions.

(3) A file with the name <macro>.CMD must exist on the disk in drive 0, and it must be in the special Macro command format.

(4) Macro commands generally do not take any parameters.

If any one of the above conditions is not met, an "Illegal command" message is issued by the Command Interpreter, and the command is ignored.

Macro command files comprise a set of recordable internal commands which are stored in a binary encoded format in order to save disk space and processing time. Since references to Variables are stored as the absolute binary addresses of these Variables and since Variable locations may change when software modifications are made, it is essential that Macro commands referring to absolute memory locations are only executed under the program version for which they were generated. A warning is issued if the user attempts to execute a Macro command which was designed for or generated by a CGCS version different from the one in use, and all Internal commands within the Macro command file which refer to absolute memory locations are dropped. (They are indicated to the operator, though, with an appropriate error message.) Macro command files generated under a previous system version have to be converted with the Macro Command Editor COMMED into a valid Macro command for the current system version.

Macro command files can be created in either of two ways:

(a) By recording actual commands during a growth run, using a Control Output file and the START and END commands, or

(b) With the Macro Command Editor COMMED which can also be used to modify command files recorded during a growth run.

The following Internal commands can be recorded on and later executed from a Macro command file:

```
CHANGE
CLEAR
DEBUG CONTINUOUSLY
DEBUG MODIFY
DEBUG OFF
DEBUG RESUME
DEBUG SUSPEND
END
IF
MODE
PLOT
RESET
SET
```

Macro commands can be invoked from a Macro command file, but they are <u>not</u> recorded in a Control Output file. This was done on purpose since a Macro command invoked from another Macro command preempts the command file from which it was invoked. (There can be only one Macro command file in use at a given time.) A Control Output file generated during a growth run receives commands issued by the operator as well as commands stemming from a Macro, and it is not possible to distinguish between both. The operator generated commands interspersed with the commands originating from the Macro would, however, be effectively lost if the Macro call were also recorded in the Control Output file. Replaying this Control Output file as a Macro file at a later stage would simply result in the Macro being preempted by the one which was invoked during the recorded run, and only the commands on the new Macro would be executed automatically. This would deteriorate the self-learning ability of the CGCS considerably.

<u>Note:</u> Commands issued by a Macro command file remain active even after the Macro was terminated or preempted!

4.6 Disk Files

Besides the Macro command (input) files, there are three files available for output from the CGCS under the operator's discretion.

PRINT FILE: The Print file receives the complete dialogue between the operator and the system. Each line of output is tagged with the absolute and the system times; the date on which the run was started and the run identification are contained in page header lines. The Print file can be opened (activated) or closed (deactivated) with the DOCUMENTATION command or via FILES. Print file output can alternatively be sent to the line printer (which is indicated by ":LP:" in the FILES display), or to a disk file. Arbitrary (valid) file names and extensions may be chosen, and the file can be opened on either disk drive. (It is recommended, though, that drive 1 is used for the Print file output because the Print file tends to become very bulky, and there is not too much room left on the system disk.) In addition to the operator dialogue, Data Dumps are recorded in the Print file which contain the following items:

* Measured values of the three heater temperatures.
* Heater power input and output values.
* Measured motor speeds.
* Seed and crucible positions.
* Crystal length and diameter.
* Weight and differential weight.
* Base temperature.
* Gas pressure.

In order to conserve space, the output items are identified only with two-character mnemonics:

```
T1 ... Heater #1 Temperature (in millivolts)
T2 ... Heater #2 Temperature (in millivolts)
T3 ... Heater #3 Temperature (in millivolts)
SL ... Seed Lift Speed (in millimeters/hour)
CL ... Crucible Lift Speed (in millimeters/hour)
L  ... Length Grown (in millimeters)
D  ... (Calculated) Diameter (in millimeters)

P1i .. Demanded Power (Input) for Heater #1 (percent)
P2i .. Demanded Power (Input) for Heater #2 (percent)
P3i .. Demanded Power (Input) for Heater #3 (percent)
SR ... Seed Rotation Speed (in RPM)
CR ... Crucible Rotation Speed (in RPM)
W  ... Crystal Weight (in grams)
DW ... Differential Weight (in grams/minute)
```

P1o .. Actual Power (Output) of Heater #1 (in percent)
P2o .. Actual Power (Output) of Heater #2 (in percent)
P3o .. Actual Power (Output) of Heater #3 (in percent)
SP ... Seed Position (in millimeters)
CP ... Crucible Position (in millimeters)
BT ... Base Temperature (in millivolts)
GP ... Gas Pressure (in PSI)

Data Dumps are initiated in the following cases:

* Upon a DUMP command.
* At a change of the system's operation mode.
* Periodically with a specifiable interval.

In the first two cases, a Data record is also written to the Data file.

DATA FILE: All important system parameters can be recorded on the Data disk file. A set of data is compiled in regular intervals and written to disk. With regard to execution time and disk space requirements, these records are written in a not directly legible binary format; special support software which can decode Data files and output selected channels, for instance, to a chart recorder, is required. The following items are contained in each data record:

Operation Mode
System Time
Length Grown
Measured Data (17 channels - all data displayed permanently)
Auxiliary Analog Data (8 channels)
Power Output (3 channels)
Current Setpoints (9 channels - all data displayed permanently)
Auxiliary Setpoints (9 channels, as above)
Debug Continuously Addresses and Data (4 * 3 channels)
Diameter
Debug Continuously Variable types (1 channel)

Each channel holds two bytes of data; one record of 64 channels (63 active, 1 spare) fills exactly one sector on the output disk.

The Data file can be opened (activated) or closed (deactivated) with the DATA command, or via FILES. Arbitrary (valid) file names and extensions may be chosen, and the file can be opened on either disk drive. (It is recommended, though, that drive 1 is used for the Data file output because the Data file

tends to become very bulky, and there is not too much room left on the system disk.) The operator has to specify an interval for the data acquisition when you open a Data file; there are about 1800 sectors available on an empty disk, and each record consumes one sector. (The remainder of the sectors on the disk is required for housekeeping.) Since it should make sense not only to record data but also to process it later on, it is probably a good idea to restrict data recording to processes which are actually of interest, and to choose the recording interval according to the dynamic behavior of the processes involved. (Once a Data file has been opened the interval can not be changed any more. A new Data file has to be opened if a different recording interval is needed.)

CONTROL OUTPUT FILE: All recordable commands (compare chapter 4.5) are recorded in a Control Output file if such a file is open, and if the START command has been issued. A Control Output file can be opened with the START command, and it can be opened and closed with FILES. The file may be opened on either disk drive, but it must be opened on drive 0 if it should serve as a Macro command file within the same run. No file name extension is required with the Control Output file name; the CGCS appends automatically ".CMD". Command recording can be deactivated with an END command at any time after a START command; the Control Output file remains open, though, until it is either closed with FILES, or until the CGCS is EXITed. One Control Output file can hold multiple Macro command records on the Control Output file which are started and terminated with the START and END commands, but the file requires editing in this case (with COMMED) before all these Macro command records can be used. (Otherwise, the first END recorded would preempt the Macro command, and all following commands would be ignored.)

Note: During a growth run, a Macro command file can be created for "instant use" in the following way:

(1) Open a Control Output file on drive 0 (important!) with an arbitrary name, preferably using the START command.

(2) Enter the command(s) you want to have in the file but be careful that you do not interfere with a growth run in progress.

(3) Close the Control Output file (with FILES), and

(4) Use it as a Macro ccmmand when required.

A Control Output file must be closed before it can be invoked it as a Macro file.

PLOT OUTPUT: In contrast to th above three output files, Plot Output is directed to an analog rather than a digital device, namely, to a multi-channel chart recorder. In general, any Variable whose type is INTEGER*2 can thus be submitted to the chart recorder output, and so can any arbitrary two-byte memory location which is referred to by its address. This includes all measured input data (which are in INTEGER*2 format anyhow), plus a number of internal system parameters. (Refer to the list of Variables in Appendix 11 to find the Variables which might be of interest.) In general, the absolute values of the Variables specified are output on the eight spare analog output channels, scaled from 0 to 10 V for the full range of 0 through 32767 of positive INTEGER*2 numbers. A message is output on the console and recorded in the Documentation output whenever a Variable changes its sign. (Initially, all outputs are supposed to refer to positive values.)

In addition to the standard INTEGER*2 Variables, the following Variables obtained from a special treatment of internal data were provided for chart recorder output:

(1) Heater and Base Temperatures: Four Variables, EXTMP1, EXTMP2, EXTMP3, and EXTMPB, hold an expanded Heater or Base Temperature value. The full range (0 to 10 V) of the output obtained from these Variables is determined by the Variables RANGT1, RANGT2, RANGT3, and RANGTB, respectively, starting from an offset value which is set by the Variables OFFST1, OFFST2, OFFST3, and OFFSTB. Like all other Variables, these parameters can be modified with the standard SET, CHANGE, or DEBUG Modify commands; their values must be specified in millivolts. In order to PLOT on the Chart Recorder Channel 3 the temperature of the Heater 1 which is supposed to lie, say, between 22.5 and 24.5 mV, the following commands may be used:

```
SET OFFST1 22.5 0
SET RANGT1 2 0
PLOT EXTMP1 3
```

Temperature values below the specified offset will result in a zero output, and values greater than the offset plus range values, in an output voltage of 10 V. Note that the offset may be ramped, too; this permits to record a deviation from a given setpoint.

(2) Growth Rate: An expanded Growth Rate value is kept in GRRATE. A zero output corresponds to a growth rate of zero (as calculated by the Diameter Evaluation routine SHAPE); the maximum output is reached for a growth rate of 20 mm/hr. GRRATE can assume positive and negative values (the latter during meltback).

(3) Diameter Error: The Variable DIAERR holds the difference between the Diameter setpoint and the actual diameter. A zero difference is output as mid-scale (5 V); zero and maximum output correspond to an actual diameter 10 mm smaller and greater than the setpoint, respectively. Greater deviations than 10 mm result in the proper minimum or maximum output signals.

(4) Crucible Position Error: Similarly, the Variable CRPERR is set to a value corresponding to the deviation of the actual crucible position from the calculated value. A zero error is again represented as mid-scale; the maximum deviation which can be resolved is ± 10 mm. (The crucible is too low if the output is less than mid-scale.)

Any PLOT channel can be activated by the command

PLOT <varname> <channel #> or
PLOT <hexaddr> <channel #>

The command may be entered in one line, or one item at a time as requested by the CGCS. The system checks whether the type of the Variable specified is indeed INTEGER*2 (it assumes INTEGER*2 locations if a hexadecimal address was entered), and attaches the value of the specified location to the proper output channel. Channel numbers 1 through 8 are permitted. An output channel remains active and connected to a Variable until it is re-assigned; output may be de-activated with the

PLOT ZERO <channel #>

command. The analog output is updated periodically once every second.

4.7 Variables

4.7.1 General Remarks

The concept of the CGCS permits an easy way to modify any arbitrary parameter used by the system, a way which is certainly more convenient and safer than using the parameter's absolute address in memory: A virtually unlimited number of parameters can be accessed by a name unique to each parameter. The CGCS looks up the actual address and the type of a specified Variable in a directory file; the number of parameters accessible in this way is only limited by the reasonably obtainable size of this file. The directory file has the name CZONAM.Vmn, with m and n, the major and minor version code numbers. It contains Variable names, addresses, and types in a binary encoded form, and is generated from a source file VARADD.SRC by means of a dedicated program CONVAD. The directory file must be updated for each new system version since the Variables listed in it may have changed their addresses due to program modifications.

Variable names must consist of one to six alphanumeric characters; the first character must be alphabetic. Variables can either be simple storage locations, or arrays. Elements of arrays must be specified by the number of the element (beginning with 1), in parentheses immediately following the array name. (There must not be a space between the name and the opening parenthesis.) An omitted array element number defaults to 1. Valid Variable names are, for example, "TIME" or "ANAPAR(6)". The name may be entered in upper- or lowercase characters.

Chapter 4.7.2 provides a list of special Variables which are more than a simple parameter since their values directly determine the operation of the CGCS. A table of the most important Variable names, sorted according to their meanings, and a complete list of <u>all</u> Variables used by the CGCS are provided in Appendix 11.

4.7.2 Special Variables

<u>System Control:</u>

TEST This Variable puts the CGCS into a Test mode if it is set to -1; all other values maintain the regular operation of the system. In Test mode, input from the A/D converter and output to the D/A converter and the relays board are inhibited. This permits to safely

assign values to an array of Variables which are otherwise set by the A/D converter's output, and to run the system with these faked "measured data" for testing purposes. (The names of the input array Variables are made up from the letter "M" plus a five character mnemonic; compare Appendix 11.) Note: TEST must not be set to -1 while the CGCS is actually controlling the puller!

DIASTA This is an internal status parameter of the Diameter Evaluation routines. It may be set to -2 at the end of a growth run in order to disable the diameter evaluation and, in particular, the generation of error messages which may be triggered by some of the actions usually involved in the close-down procedure of the puller. Diameter evaluation may be enabled again with a RESET command.

ALPHA The parameter ALPHA determines the diameter evaluation algorithms within two extreme approaches. ALPHA should be a floating-point number between 0 and 1. For further information, see chapters 4.1.3 and 5.3.2.2.3).

XTLSHP This parameter holds (in floating-point format) the maximum permitted difference between the squares of the diameter of the crystal (in millimeters) in two adjacent sections of the crystal, approximately 1.2 millimeters apart from one another. The square of the diameter stored for buoyancy compensation purposes is adjusted, if necessary, to differ by not more than the value of XTLSHP from the preceding value.

Display Control:

INTRVL This Variable determines the duration of the intervals between subsequent output operations to the console. One unit corresponds to an interval of 50 milliseconds. The default value of 10 corresponds to a complete screen update every four to six seconds, depending on the other activities within the CGCS. More frequent updates may be required during testing and alignment; they can be achieved with smaller INTRVL values. The fastest screen update is done with INTRVL set to 1; a zero INTRVL value disables the screen output entirely. Note: The screen display will "freeze" irreversibly if INTRVL is set to zero; regular operation will not be resumed even if INTRVL is set back to a non-zero value. The system has to be

restarted in order to re-activate data output on the screen. (The CGCS remains operable, though, with the screen output disabled.) INTRVL does not affect the output of the time, of operator commands, and of system messages.

Data Dump Control:

DUMPIN The Variable DUMPIN holds the interval between periodical Data Dumps to the Print file; the time units are minutes. DUMPIN may be set to any convenient value at any time; a DUMPIN value of zero disables the periodical Data Dumps.

DUMPFL This Variable triggers an additional Data Dump (and an additional record written to the Data file) if it is set to -1. Note that a SET DUMPFL -1 0 command is the only save way to trigger additional Data Dumps from a Macro Command file. (DUMPFL is reset by the Data Dump routine; it must therefore be set to -1 repeatedly if more than one Data Dumps are required.)

Scratchpad Variables:

DUMMY In order to facilitate advanced Macro programming, eight dummy INTEGER*2 locations were provided. These locations are not accessed by the CGCS code proper, but they may be arbitrarily ramped or used as flags (set to specific values) and employed in Conditional Macro commands. The dummy locations are referred to as DUMMY(1) through DUMMY(8).

Miscellaneous - Read-Only Variables:

TIME The Variable TIME holds the current system time (in seconds) in an unsigned two-byte INTEGER location. This counter wraps around to zero after 65,536 seconds. Note that the contents of TIME are interpreted as a signed INTEGER*2 number by the display and also by the Conditional Macro Command execution routines; time counts greater than 32,767 seconds are thus interpreted as negative numbers.

RAMPNG This Variable holds the number of parameters which are currently being ramped. You may look at it (and have your Macro commands look at it), but messing around

with RAMPNG will inevitably confuse the CGCS. The results may be spectacular but probably not desirable.

CNDCNT The same considerations as to RAMPNG apply to the count of pending Conditional Macro commands kept in this Variable.

ZERO This location holds, simply enough, a zero INTEGER*2 value. You may try to modify it but you won't be very successful since this location is in ROM and thus inaccessible to any writing attempt.

5. The Czochralski Growth Control System Software

5.1 CGCS Concept and Structure

5.1.1 Program Structure

From the programmer's point of view, the Czochralski Growth Control System (CGCS) is an iRMX-80 based real-time application system consisting of a number of iRMX-80 "tasks". A task is a section of program code, usually dedicated to one control commission or part of it. It is more or less independent from other tasks and is executed whenever its specific action is required and system resources are available, according to the priority level which has been assigned to it. The execution of a task is scheduled by the operating system's "Nucleus", either in response to extraneous events (interrupts), or when a task receives data which it was waiting for in the form of a "message" from a fellow task.

From the user's point of view, however, the CGCS consists essentially of three functional groups each of which, in turn, consists of several tasks:

(1) The System Interface: This part of the software is transparent to the user. It provides, nevertheless, essential functions like data formatting, input and output, or timekeeping.

(2) The Operator Interface: These tasks form the link between the operator and the controller routines proper. Holding the system's "intelligence", they constitute the by far largest part of the CGCS code. The Operator Interface is responsible for the following actions:

   (a) Prompting for and interpretation of operator commands which control the functions of the CGCS.

   (b) Execution of operator and Macro command file sourced commands. This function was kept strictly separate from the operator command interpretation in order to facilitate the handling of Macro commands.

   (c) Recording of all commands pertaining to the actual crystal growth process.

   (d) Periodic output of measured data on the console CRT terminal, and to a disk file, and preparation of data to be output on an analog chart recorder.

(3) The Process Controller proper: These are the routines actually involved in controlling the heater power(s) and motor speeds according to the pertinent setpoints provided by the Operator Interface. They also constitute the interface to the analog and digital I/O hardware.

We will follow the above scheme for the subsequent discussion of the CGCS software. Chapter 5.2 is devoted to the large number of system and system interface routines which, due to their rather generic design, can be regarded as "black boxes" within the controller code proper. The actual controller code is discussed in chapter 5.3, in two sub-sections corresponding to the operator interface, and the process controller, respectively.

5.1.2 General Program Information

The CGCS consists of routines part of which were written in FORTRAN, part in assembly language. In general, the operator interface and part of the actual controller routines are FORTRAN-based, whereas the system interface modules (and all system routines which were not supplied by Intel) are coded in assembly language. Assembly language was chosen when one or more of the following requirements had to be met:

* Interface to iRMX-80 system routines which cannot be called directly from FORTRAN due to different parameter passing conventions.

* High operation speed, which is particularly important if a routine is invoked very frequently.

* Numeric operations which can be coded more efficiently in assembly language than in FORTRAN (e.g., the low-pass filtering algorithm).

FORTRAN, on the other hand, was chosen where the use of a high-level language was considered advantageous with regard to program clarity and programming efficiency. It was the obvious choice for routines which involve floating-point arithmetics. In order to improve the execution speed and code efficiency of FORTRAN, a set of library routines was implemented which replace the standard (lengthy and slow) FORTRAN floating-point algorithms by routines which make use of the 8231 Numeric Processor. These routines are not only several kilobytes smaller than the standard ones, they also boost the execution speed by about one order of magnitude.

A special approach was necessary to fit the CGCS into the available memory of less than 54 KBytes. (More than 10 of the total 64 KBytes are required for the ROM resident system and its data areas in RAM.) The entire code of the Czochralski system would have exceeded this limit by far. It was, therefore, necessary to choose an overlay approach (Fig. 15): Program code which is not required permanently within the system is loaded into a reserved memory area only when needed, overwriting an other currently dispensable overlay. The only function where this is possible without unduly impeding the system operation is the Command Interpreter which controls the dialogue between the operator and the system. Since the operator can only enter one command at a time, and since human command entry is a very slow procedure anyway, compared to the standards of a microcomputer, it was possible to split the Command Interpreter's functions into a total of 22 different overlays each of which is in charge of one particular command or a group of related commands. According to the size of the largest overlay, a memory area of 2 KBytes was reserved for the Command Interpreter overlays; the total combined size of all overlays is approximately 30 KBytes.

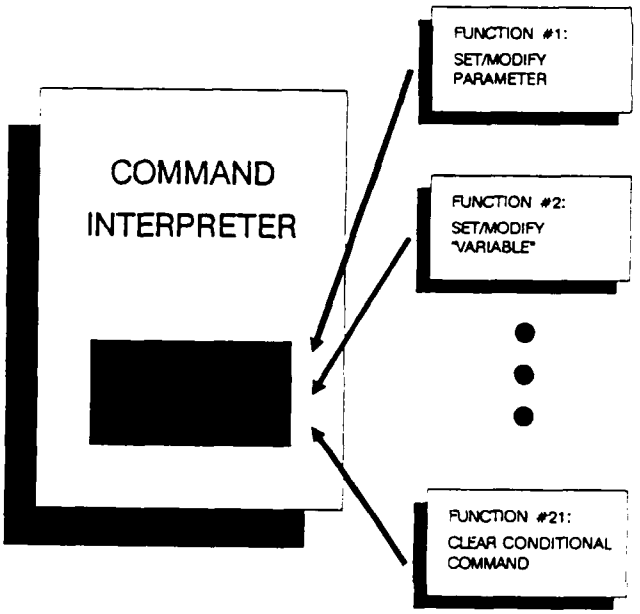


Fig. 15: Command Interpreter overlays.

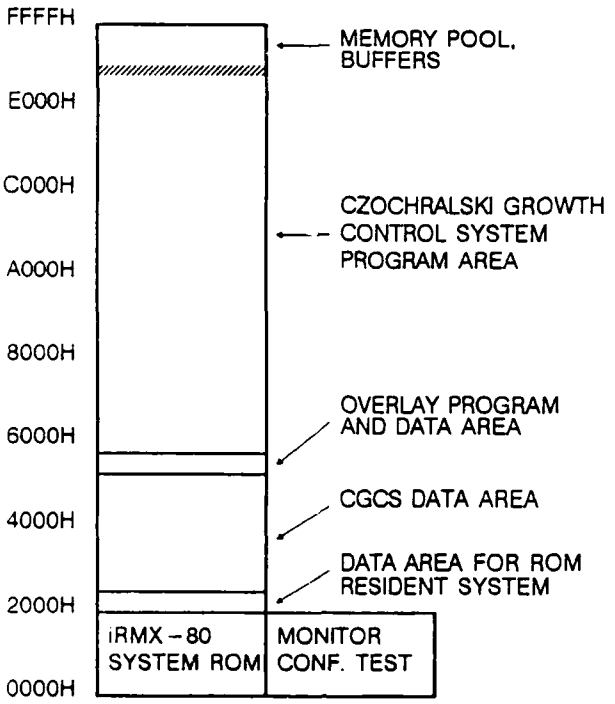


Fig. 16: **Memory map of the CGCS.**

The layout of the Czochralski Growth Control System memory map (Fig. 16 and Appendix 7) was chosen to facilitate software updating. The Variable concept for an easy modification of internal system parameters (compare chapters 1.3 and 4.7) requires a translation table which correlates the symbolic name of a system "Variable" to its physical storage location in memory. Since this translation table has to be generated manually, it is obviously not desirable if it has to be rewritten totally after each minor modification of the controller software. The system grows or shrinks at its high-address end; therefore, all important system Variables were located at the lowest addresses available, immediately above the code and data areas of the ROM resident system, in order to prevent them from being affected by system size changes. Most of these data must be available to several system tasks; extensive use was therefore made of named FORTRAN "COMMON" blocks which are arranged (in alphabetical order) at the lowest addresses and consume approximately 1,280 bytes. (Since FORTRAN COMMON blocks require a special treatment at program linkage time in the Intel 8080/85 environment, it is

again advantageous to have them all located at addresses which are least liable to change.) The COMMON blocks are immediately followed by the general system data area. The lowest addresses within this area are used by the data locations of assembly language modules some of which have to be manually "tied" to "COMMON" blocks; these locations are still not very likely to be affected by program modifications. They are followed by the data areas of the permanently resident FORTRAN based software which are essentially scratchpad locations for the internal use of these routines. The remainder of the data area whose total size is approximately 9,900 bytes holds system data which hardly need be explicitly accessed and whose actual absolute addresses do, therefore, not matter.

A 2 KByte range immediately above the data area is reserved for the Command Interpreter overlays' code and local data. It is succeeded by the bulk of the system code. This code area has currently a size of about 39.5 KBytes; the area between its top and some disk buffers and system variables which reside next to the high-address end of RAM is used as a memory pool from which memory can be dynamically assigned to system tasks when required. The size of this memory pool does not matter unless it becomes too small; the program code may therefore grow without penalty due to software improvements. (The memory reserves are currently in the order of 1.5 KBytes, which does permit program improvements but certainly not the introduction of major new features.)

5.2 System Interface and Auxiliary Routines

The routines listed in this section are of a rather generic nature. Although they have been initially developed for a process controller similar to the CGCS (and considerably improved since), their supporting nature distinguishes them from the genuine process controller software which will be discussed in chapter 5.3. In general, these routines can be regarded as "black boxes" as far as the CGCS is concerned; some details about their operation and their interface to the software from which they are called are presented here, however, in order to permit a more thorough understanding of the CGCS software proper.

The system interface and auxiliary routines are, in general, kept in various libraries from where they are linked with the actual CGCS software when required. The following libraries are used within the CGCS:

FRXMOD.LIB contains all Fortran-iRMX-80 interface, access, and data transfer control routines. These modules can only be executed in an iRMX-80 environment.

FIORMX.LIB is the I/O formatting library for execution under iRMX-80.

FORTIO.LIB interfaces the I/O conversion routines to a FORTRAN environment.

FXDISK.LIB permits to perform directory-controlled disk or device I/O under RMX-80.

FXUTIL.LIB comprises a set of auxiliary utility routines which may or may not require interface routines contained in the above libraries.

Similar routines for an ISIS-II or RXISIS-II environment are used by auxiliary programs supporting the CGCS:

FIOISS.LIB holds the (slightly simplified) ISIS-II versions of the routines in FIORMX.LIB.

FIORXI.LIB offers all features of the FIORMX.LIB but can be executed with less overhead under RXISIS-II.

FXDSKI.LIB is equivalent to FXDISK.LIB for an ISIS-II or RXISIS-II environment.

5.2.1 iRMX-80 Control Routines - Library FRXMOD.LIB

NAME	TYPE	FUNCTION	CHAPTER
FXSEND	subr	non-reentrant msg. sending rout.	5.2.1.1
FXWAIT	subr	non-reentr. msg. receiving rout.	
FXACPT	subr	non-reentr. msg. receiving rout.	
FRSEND	subr	reentrant message sending rout.	5.2.1.2
FRWAIT	subr	reentr. message receiving rout.	
FRACPT	subr	reentr. message receiving rout.	
FRINIT	subr	initialization routine	
FRCRSP	func	check for response message	
FRCXCH	subr	exchange creation routine	5.2.1.3
FRDLVL	subr	interrupt level disabling rout.	
FRDTSK	subr	task deletion routine	
FRDXCH	subr	exchange deletion routine	
FRELVL	subr	interrupt level enabling routine	
FRRESM	subr	task execution resuming routine	
FRSUSP	subr	task execution suspending rout.	
FRACTV	func	task descriptor of running task	
FXCFLG	task	flag interrupt creation task	5.2.1.4
FXCRFE	subr	create flag interrupt exchange	
FXDLFE	subr	disable flag interrupt exchange	
FRACCS	subr	access common resources	5.2.1.5
FRRELS	subr	release common resources	
FRINAR	subr	create an access control exch.	
FXSYSE	subr	system error reporting routine	5.2.1.6
FRIFSM	subr	Free Space Manager initializ.	5.2.1.7

5.2.1.1 Non-Reentrant Message Sending/Receiving Routines

The three routines FXSEND, FXWAIT, and FXACPT permit the transmission and reception of messages with arbitrary lengths. They can be called as subroutines by a FORTRAN program. Message data are physically located in memory supplied by the iRMX-80 Free Space Manager. FXSEND builds a message within these memory locations, copying the data indicated by the "variable" and "length" parameters to the message. Therefore, the sending task may change the data which was submitted to FXSEND immediately after the call for FXSEND. The data which

is to be included into the message must, however, be located in contiguous memory locations (compare chapter 3.1.5.3). FXSEND transfers the number of bytes specified by the "length" parameter, starting with the location indicated by "variable". "Variable" is therefore the name of the first variable (independent of its type) within the data block. It remains in the responsibility of the programmer to specify a correct byte count with "length" as there is no possibility whatsoever for FXSEND to check for the actual data block length. A zero data string length is permissible; still, a (dummy) "variable" name must be specified even in this case. Note: The maximum permitted "length" value for FXSEND is 243; larger values cause a "SYSTEM ERROR" message at execution time, and the "send" command is ignored.

The message dispatched by FXSEND may be received by any message receiving routine described in this or in the next chapter. The routines FRWAIT and FRACPT will return it to the response exchange which was specified with the FXSEND call, after having copied the data sent with the message to memory locations of the receiving task. A correct response exchange must therefore be specified with the FXSEND call if a task might use FRWAIT or FRACPT in order to receive the message at the specified exchange. The routines FXWAIT and FXACPT, on the other hand, return the memory used for building the message to the free space manager, and no response message is generated. In this case, the "response exchange" parameter in the FXSEND call may be any dummy variable name; it must, however, not be omitted. Anyhow, the "life" of the message sent via FXSEND must be terminated either by the receiving task or by any task which services the response exchange (if one was specified) with FXWAIT or FXACPT.

The functions of FXWAIT and FXACPT correspond to those of the iRMX-80 system routines RQWAIT and RQACPT, respectively. A task which performs an FXWAIT call waits at the exchange specified with the call either until a message is available at this exchange or until the time limit (if requested) is over. If a task times out at an exchange, FXWAIT sets the "length" parameter to zero and returns a "type" value of 3 (TIMED$OUT-$MSG) if a "type" value of zero has been specified in the FXWAIT call. Note: The parameter "time limit" must indicate an INTEGER*2 variable! This demand is automatically fulfilled if numeric (integer type!) constants are used in conjunction with the default integer length of the FORTRAN compiler FORT80.

FXACPT, on the other hand, checks whether a message is available at the specified exchange. If so, the message is removed

from the exchange and processed. If there is no message, FXACPT returns a "length" value of zero.

An untimed or prolonged wait performed with FXWAIT does not contradict the demands for non-reentrant interlock protected routines: The first part of the FXWAIT and FXACPT code is made reentrant, permitting an unlimited quasi-parallel use of this code by an arbitrary number of routines. Only the last part of these routines - the returning of the memory used for the message to the Free Space Manager - had to be made non-reentrant.

The further treatment of the message is the same for FXWAIT and FXACPT: The routines first check the "type" value specified with their call. If FXWAIT or FXACPT were called with a "type" parameter of zero, a message of any type is accepted; the message "type" value is copied to the FXWAIT or FXACPT "type" parameter. For any non-zero "type" parameter, FXWAIT and FXACPT check the message type value. If it is equal to the specified value, the message is further processed, otherwise, a "SYSTEM ERROR" message is issued, and FXWAIT or FXACPT try to receive another message at the same exchange. A zero "type" value must therefore be used if messages with different "type" values may be received; a further check can be performed by the receiving task. The type checking feature, on the other hand, permits the detection of misguided or erroneous messages.

Having accepted the message as correct, the number of bytes which has been specified with the "length" parameter when the message was generated by the sending task is copied from the data area of the message to a data area in the receiving task's memory which is defined by the parameter "variable". Therefore, the data pattern which existed in the data block of the sending task when the message was built is copied to a data block within the receiving task. This data block starts with the location indicated by "variable", as explained for FXSEND. The number of received data bytes is returned in the "length" parameter. Note that FXWAIT and FXACPT return only one "length" byte. The variable specified for "length" should therefore either be declared as INTEGER*1 or explicitly set to zero prior to the FXWAIT or FXACPT call. Otherwise, accidental data in the high byte(s) of an INTEGER*2 or *4 variable would cause a totally meaningless value. Furthermore, the programmer has to make sure that the allocation of the message data within the receiving task corresponds to the allocation within the sending task, i.e., the number, types, and order of the variables within the data block of the receiving task must be the same as in the sending task.

Finally, FXWAIT and FXACPT check whether the message was actually generated in memory supplied by the Free Space Manager. If so, the message is returned to the memory pool of the Free Space Manager. No further action is taken if the message has not been created by FXSEND (and was therefore not built from Free Space Manager memory). This permits the receiving of any arbitrary message by FXWAIT or FXACPT.

ROUTINE FXSEND:

Routine Type: Assembly language subroutine; not reentrant; protected by a software interlock.

Initialization: Execution of FXITSK.

Routine Call:

```
CALL FXSEND (receiv.ex.,resp.ex,variable,length,type)
```

with: receiv.ex.: Name of the exchange to which the message is sent.
resp.ex: Name of the response exchange to which the message should be returned.
variable: Name of the first variable in a contiguous data block which is to be transmitted.
length: Number of bytes to be transmitted (or name of an INTEGER*1 or INTEGER*2 variable holding this value).
type: Message type value (or name of an INTEGER*1 or INTEGER*2 variable holding this value).

Required Stack: 10 bytes

ROUTINE FXWAIT:

Routine Type: Assembly language subroutine; not reentrant; protected by a software interlock.

Initialization: Execution of FXITSK.

Routine Call:

```
CALL FXWAIT (exchange,time lim.,variable,length,type)
```

with: exchange: Name of the exchange where the task is to wait for a message.

time lim.: Time limit (INTEGER*2 constant or name of an INTEGER*2 variable holding this value).
variable: Name of the first variable in a contiguous data block which is to be updated by the message.
length: Name of an INTEGER*1 variable where FXWAIT stores the number of data bytes received.
type: Type value or name of an INTEGER*1 or INTEGER*2 variable where FXWAIT stores the "type" value of the received message if a zero value has been specified.

Required Stack: 18 bytes

ROUTINE FXACPT:

Routine Type: Assembly language subroutine; not reentrant; protected by a software interlock.

Initialization: Execution of FXITSK.

Routine Call:

```
CALL FXACPT (exchange,variable,length,type)
```

Parameters: see FXWAIT

Required Stack: 18 bytes

5.2.1.2 Reentrant Message Sending/Receiving Routines

In order to avoid the unpredictable execution delays which may be imposed upon a task using FXSEND, FXWAIT, or FXACPT, a second set of message transmitting and receiving routines was provided. While a task using one of the above routines might have to wait at the interlock exchange until the routine becomes available and would probably incur some further delay during the memory allocation performed by the Free Space Manager, there is no (inherent) delay if the reentrant routines FRSEND, FRWAIT, and FRACPT are used. The characteristics of these routines, however, differ slightly from those of the non-reentrant ones.

The major difference is caused by the fact that FRSEND does not copy the data which is to be dispatched. In contrast, FRSEND uses a fixed message which is closely connected to a dedicated response exchange. No message must be explicitly

sent to this exchange. The response exchange and the message header form a 19 byte block in the data area of the transmitting task. This block must immediately precede the block of data which is to be sent with the message. (This can be guaranteed as shown in chapter 3.1.5.3 if the first variable to be transmitted is preceded by a 19-element INTEGER*1 array whose name is the name of the response exchange.) Note that the name of the response exchange need not be made public to other tasks as the message itself contains the corresponding information. Prior to the first call for FRSEND, the transmitting task must call the routine FRINIT which creates the response exchange and initializes the message header. Note: FRINIT must be called once and only once by each task which is going to use FRSEND. The meaning of the parameters in the FRINIT call is similar to the FXSEND parameter set. The data block length is limited to 246 for the FRSEND routine; a larger value will cause a "SYSTEM ERROR" message, and the task which has issued the FRINIT call is effectively suspended (waiting forever at its own response exchange). As all information about the message length and type has already been defined in the FRINIT call, the set of parameters required for the FRSEND call differs significantly from the one required with FXSEND.

A very important advantage of the fixed message locations used with FRSEND is that the actual working data area of a task may be transmitted to another task without the need of copying it within the sending task. Still, this imposes a problem as any changes of the data within the message are legal only after the message was received, copied, and returned by the receiving task. Variables within the message data block should therefore be changed only when the message is waiting at its response exchange. Before writing data into the message data block, the user task should check the LOGICAL*1 function FRCRSP which returns a .FALSE. value if the (correct) message is waiting at the response exchange, and otherwise a .TRUE. value. If FRCRSP returns a .TRUE. it is suggested to skip the message preparation and dispatch entirely as the receiving task would not be ready to accept new data anyhow. This applies, of course, particularly if the transmitting task runs periodically in order to update its output data. Only one FRCRSP check is required before each message dispatch; once the message has been returned to its response exchange, it can only be removed from there by an FRSEND call. FRSEND, in turn, checks independently from FRCRSP whether the message is actually available for sending, and removes it from the response exchange. If no message (or not the correct message) was waiting at the response exchange, FRSEND returns to the calling task without further notice and without having sent a message. This implies, however, another important conse-

quence: a task cannot perform several succeeding FRSEND calls with the same message if its priority is higher than the priorities of the receiving tasks. In this case, the sending task continues running and does not permit the first receiving task to remove the message from its input exchange. All other tasks would therefore forever be locked out from data transfer.

While the exchanges to which the non-reentrant routine FXSEND sends its messages may be served by FXWAIT and FXACPT as well as by FRWAIT and FRACPT, this does not apply analogously to the FRSEND messages. They must be received either by FRWAIT or by FRACPT. Only these routines return a message to its response exchange. Note that any message, no matter what its origin was, is sent to the location indicated by its response exchange field if its length exceeds eight bytes. (This message length is not identical with the parameter "length". The actual message length results from the value of "length" plus nine (for the message header).)

Aside from the final treatment of the message, the characteristics of FRWAIT and FRACPT are identical to those of FXWAIT and FXACPT. FXWAIT and FXACPT return the message to the Free Space Manager after having processed it; their reentrant counterparts send it back to the response exchange. The information given about FXWAIT and FXACPT in chapter 5.2.1.1 applies therefore analogously to FRWAIT and FRACPT, respectively.

ROUTINE FRSEND:

Routine Type: Assembly language subroutine; reentrant.

Initialization: Call for FRINIT.

Routine Call:

```
CALL FRSEND (receiv.ex., resp.ex.)
```

with: receiv.ex.: Name of the exchange to which the message is sent.
resp.ex: Name of the response exchange to which the message should be returned.

Required Stack: 10 bytes

ROUTINE FRWAIT:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none.

Routine Call:

```
CALL FRWAIT (exchange,time lim.,variable,length,type)
```

with: exchange: Name of the exchange where the task is to wait for a message.
time lim.: Time limit (INTEGER*2 constant or name of an INTEGER*2 variable holding this value).
variable: Name of the first variable in a contiguous data block which is to be updated by the message.
length: Name of an INTEGER*1 variable where FRWAIT stores the number of data bytes received.
type: Type value or name of an INTEGER*1 or INTEGER*2 variable where FRWAIT stores the "type" value of the received message if a zero value has been specified.

Required Stack: 18 bytes

ROUTINE FRACPT:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none.

Routine Call:

```
CALL FRACPT (exchange,variable,length,type)
```

Parameters: see FRWAIT

Required Stack: 18 bytes

ROUTINE FRINIT:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none.

Routine Call:

```
CALL FRINIT (resp.ex,length,type)
```

with: resp.ex.: Name of the 19 byte response exchange - message header block immediately preceding the data block.
length: Number of bytes to be transmitted (or name of an INTEGER*1 or INTEGER*2 variable holding this value).
type: Message type value (or name of an INTEGER*1 or INTEGER*2 variable holding this value).

Required Stack: 12 bytes

ROUTINE FRCRSP:

Routine Type: Assembly language subroutine; reentrant; must be declared as LOGICAL*1 in the calling FORTRAN program.

Initialization: none

Routine Call:

```
boolean = FRCRSP (resp.ex.)
```

with: boolean: LOGICAL*1 variable (or immediate use of FRCRSP as parameter, e.g., in a logical IF statement)
resp.ex: Name of the response exchange which is tested for the waiting message.

The routine returns .TRUE. if the correct response message is not waiting at the specified exchange, and .FALSE., if it is waiting.

Required Stack: 0 bytes

5.2.1.3 Interface Routines for iRMX-80 Nucleus Functions

The remaining iRMX-80 Nucleus routines - as far as they are applicable to a FORTRAN based system running on hardware such as an iSBC 80/24 board - are interfaced by the routines described within this chapter. (No interface routine was provided for the iRMX-80 task creation routine RQCTSK.) Since these interface routines simply adapt the parameters supplied by FORTRAN to the requirements of iRMX-80, they maintain completely the characteristics of the corresponding iRMX-80 sys-

tem routines (whose names result from the routine names if the first two characters "FR..." are replaced by "RQ..."). All routines specified within this chapter are reentrant.

Two routines are provided for the creation and deletion of exchanges, FRCXCH and FRDXCH, respectively. The exchange address parameter of both routines must specify a ten byte location in memory. FRCXCH builds and initializes an exchange in any case. In contrast, FRDXCH checks first whether a task or a message is waiting at the specified exchange. If so, no further action takes place, and FRDXCH returns a .FALSE. value to the variable specified as its second parameter. If neither a task nor a message is waiting at the exchange, the exchange is deleted, and FRDXCH returns a .TRUE. value.

Three routines permit the control of the status of a task: It can be deleted (with FRDTSK), suspended (with FRSUSP), or its execution can be resumed if it was suspended (with FRRESM). These three routines require the name of an INTEGER*2 variable as a parameter which holds the address of the task's task descriptor. There are two possibilities for supplying task descriptor addresses to a FORTRAN program: They may either be stored in a (named) COMMON block by a small assembly language or PL/M routine which may be called as part of the initialization sequence, or each task determines its own task descriptor address by calling the (INTEGER*2) function FRACTV, and stores the task descriptor address returned by FRACTV in memory.

Two routines, FRELVL and FRDLVL, finally, permit the enabling and disabling of interrupt levels, respectively. The appropriate interrupt level must be specified as a parameter with their call.

ROUTINE FRCXCH:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine Call:

```
CALL FRCXCH (exchange)
```

with: exchange: Name of a 10 byte area in read-write memory where iRMX-80 can build an exchange.

Required Stack: 2 bytes

ROUTINE FRDLVL:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine Call:

```
CALL FRDLVL (level)
```

with: level: Interrupt level (see iRMX-80 documentation) constant or INTEGER*1 or INTEGER*2 variable name holding this value.

Required Stack: 2 bytes

ROUTINE FRDTSK:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine Call:

```
CALL FRDTSK (task descriptor)
```

with: task descriptor: Name of an INTEGER*2 variable holding the address of the task descriptor of the task to be deleted, or FRACTV function call.

Required Stack: 4 bytes

ROUTINE FRDXCH:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine Call:

```
CALL FRDXCH (exchange,boolean)
```

with: exchange: Name of the exchange to be deleted.

boolean: Name of a LOGICAL*1 variable whose value is returned by FRDXCH depending on whether the

exchange could be deleted (.TRUE.) or not (.FALSE.).

Required Stack: 6 bytes

ROUTINE FRELVL:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine Call:

```
CALL FRELVL (level)
```

with: level: Interrupt level (see iRMX-80 documentation) constant or INTEGER*1 or INTEGER*2 variable name holding this value.

Required Stack: 2 bytes

ROUTINE FRRESM:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine Call:

```
CALL FRRESM (task descriptor)
```

with: task descriptor: Name of an INTEGER*2 variable holding the address of the task descriptor of the task to be resumed.

Required Stack: 4 bytes

ROUTINE FRSUSP:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine Call:

```
CALL FRSUSP (task descriptor)
```

with: task descriptor: Name of an INTEGER*2 variable holding the address of the task descriptor of the task to be suspended, or FRACTV function call.

Required Stack: 4 bytes

ROUTINE FRACTV:

Routine Type: Assembly language subroutine; reentrant; must be declared as INTEGER*2 in the calling FORTRAN program

Initialization: none

Routine Call:

variable = FRACTV (dummy)

with: variable: Name of the variable where the task descriptor address of the running task can be stored.
dummy: Name of a dummy variable which may be of any arbitrary type except CHARACTER.

The routine returns the start address of the task descriptor of the running task as an INTEGER*2 variable.

Required Stack: 0 bytes

5.2.1.4 "Flag Interrupt" Service Routines

A special feature called "flag interrupts" allows a task to indicate to another task the occurrence of an event (e.g., of a clock tick) without the overhead inherently involved in sending a message. Instead of dispatching a message, the "transmitting" task sets a one-byte "flag" location in memory to 0FFH. Each flag location is linked to a message-exchange pair similar to iRMX-80 interrupt exchanges. A dedicated task runs periodically every iRMX-80 clock tick (50 ms), checking all flag locations of whose existence it has been notified, sending the "flag interrupt" message to the corresponding exchange if it finds a flag set, and resetting all flags to zero. The execution of any other task can thus be controlled by the flag status if the task is to wait at the flag interrupt exchange. Although this approach inherently causes a delay between the setting of the flag and the processing of

the flag interrupt, it reduces the overhead for the transmitting task significantly (since modifying one byte in memory is obviously faster than executing all the iRMX-80 Nucleus operations involved in sending a message), which may be important for tasks with critical timing.

The software provided for the servicing of "flag interrupts" consists of one task and two non-reentrant routines which may be called by any task in order to create or delete a "flag interrupt" exchange. The task, FXCFLG, runs once each system time unit (50 ms) and polls all flag bytes which have been previously specified to it by calling the "flag interrupt" exchange creation routine FXCRFE. If a flag byte is found set, it is reset, and a message adjacent to the "flag interrupt" exchange is sent to this exchange. If a message is already waiting at this exchange, FXCFLG changes only the "type" byte of the interrupt message from its normal value of 1 (INT$TYPE) to 2 (MISSED$INT$TYPE) in order to indicate to the task(s) which service(s) the exchange that at least one "flag interrupt" has been missed. Note that FXCFLG does not check whether the message waiting at the exchange is actually the pertinent interrupt message; no other messages should therefore be sent to a "flag interrupt" exchange.

The priority of FXCFLG should be set rather high, in any case higher than the priorities of the tasks which might use "flag interrupts". It might be even necessary to assign a priority to FXCFLG which is (numerically) smaller than 128, i.e., a priority in the range used by the genuine interrupt service routines.

A task may receive the information that a "flag interrupt" has happened by simply waiting at the "flag interrupt" exchange in an untimed wait. Either FXWAIT or FRWAIT may be used for this purpose. Having received the interrupt message, the task may check its "type" byte in order to make sure that no "flag interrupt" was missed. (Exactly the same proceedings are required for tasks acknowledging genuine interrupts.)

Having terminated the flag byte polling loop, FXCFLG checks whether there was a request for creating or deleting a "flag interrupt" exchange. If there was one, it is executed before FXCFLG returns to its timed wait. The exchange FXCDFE to which such requests are sent by FXCRFE and FXDLFE must be initialized by the configuration module in order to guarantee its existence when the first message is sent to it.

FXCFLG keeps its pointers to flag byte and exchange locations in memory supplied by the Free Space Manager. Therefore, flag bytes and the corresponding exchanges may be dynamically

introduced by the other tasks in the system. For each "flag interrupt", FXCFLG requests eight bytes from the Free Space Manager which hold the flag byte and the exchange addresses and a pointer to the next eight-byte block which may or may not be contiguous to the preceding one. (FXCFLG uses only six of the eight bytes; eight bytes, however, are the smallest amount of memory which can be allocated by the Free Space Manager.) A newly created "flag interrupt" control block is added as the first block to be checked within a polling cycle; this reduces the program overhead considerably. In order to delete a "flag interrupt" exchange upon a corresponding request, FXCFLG searches for the specified control block, changes the pointer of the preceding block in order to thread it to the block following the one to be deleted, and returns the memory block to the Free Space Manager.

Two non-reentrant subroutines, FXCRFE and FXDLFE, permit the creation and the deletion of "flag interrupt" control structures. Both routines use the routine FXSEND in order to send an appropriate message to FXCFLG. In order to create a "flag interrupt" exchange, the name of a 15 byte location in RAM must be specified where FXCFLG can build an interrupt exchange. The "flag interrupt" exchange starts its operation immediately after the FXCRFE call. The deletion of "flag interrupt" control structures is, in contrast, a somewhat more complicated procedure. First, the task which wants to delete the interrupt exchange should call FXDLFE which disables the control structures maintained by FXCFLG and prevents any future "flag interrupts". Due to the possible delay between the deletion request and the actual deletion of the FXCFLG control block, there might be still the possibility of a "flag interrupt" after the FXDLFE request was executed. The task performing the deletion should therefore incur a timed wait (with FRWAIT) at the "flag interrupt" exchange which should last at least one time unit, better, several time units. Having made sure thus that no "flag interrupt" is to happen any more, the task may delete the interrupt exchange (with FRDXCH). Keep in mind that a message sent to a non- (or no more) existing exchange may cause a disastrous system error!

TASK NAME:	FXCFLG
ENTRY POINT:	FXCFLG
STACK LENGTH:	36 bytes
PRIORITY:	≈ 128 (higher than all tasks using its services)
DEFAULT EXCH.:	none
EXTRA:	0
INITIAL EXCH.:	FXCDFE

ROUTINE FXCRFE:

Routine Type: Assembly language subroutine; not reentrant; protected by a software interlock.

Initialization: Execution of FXITSK

Routine Call:

```
CALL FXCRFE (exchange,flag)
```

with: exchange: Name of a 15 byte location in read-write memory where the "flag interrupt" exchange can be created.
flag: Name of a flag byte.

Required Stack: 24 bytes

ROUTINE FXDLFE:

Routine Type: Assembly language subroutine; not reentrant; protected by a software interlock.

Initialization: Execution of FXITSK

Routine Call:

```
CALL FXDLFE (exchange)
```

with: exchange: Name of the "flag interrupt" exchange to be deleted.

Required Stack: 22 bytes

5.2.1.5 Access Control Routines

Three routines permit to establish and maintain software interlocks for common code or data. The routine FRINAR builds a message-exchange combination in 15 bytes of contiguous memory, and allows access to the protected sequence by sending the message to the exchange. FRACCS, in turn, performs a "wait" operation at the specified exchange. If the release message is available at the exchange, it is removed, and the task which has called FRACCS can continue its execution. Otherwise, the task has to wait until the task which is currently using the protected resources has terminated its execution and sent the release message back to the control exchange, calling

FRRELS. FRACCS checks whether the release message was the correct one; if not, a "SYSTEM ERROR" message is generated, and FRACCS continues waiting for the correct release message.

This implies several important rules for the use of these routines:

First, the programmer has to make sure that the control exchange is already created at the time the first task wants to access it. This can only be done by calling FRINAR in an initialization routine, once for each exchange-message combination. The configuration module must not be used for this purpose as it would only create the exchange without sending the message to it, which would, of course, block all tasks which would wait at the exchange. Defining the exchange in the configuration module and executing FRINAR at a later stage would be even worse: a fatal system error might happen if a task was already waiting at the exchange when the FRINAR call was issued.

Second, the sequence which is enclosed by FRACCS and FRRELS has to be kept as short as possible. Although the three routines described in this chapter are reentrant, their execution might affect the regular scheduling of the iRMX-80 tasks (compare chapter 3.1.3). The probability that this might matter is the higher the longer a task remains within a protected area and the more tasks want to access this protected area. Separate access control exchanges should therefore be provided for each independent unit (code or data) which may be used by several tasks. FRACCS should be called immediately before accessing the shared resources, and FRRELS, immediately after having left them. The shared routines ought not to perform actions which might lead to additional delays of their execution: an untimed wait, for example, would not only affect the task currently executing within the common code but also all other tasks which might want to access it. If the software interlock is used to protect data in common blocks, each task accessing these data should only copy them to or from local memory locations under the protection of the access control routines; any further operation should be done by code outside the protected sequence.

Third, one but only one FRRELS call must follow an FRACCS call. If a protected routine branches, each branch must be terminated by FRRELS; the same applies analogously to routines with several entry points. Omitting an FRRELS call after having exited the protected code would not only lock out the common code or data forever, it would also lock out all routines which would ever attempt to access it. A surplus FRRELS

call, in contrast, is ignored due to the special structure of this routine.

Note: No message must ever be sent to the control exchange except by FRRELS. The control exchange must not serve any other purpose but controlling the access to the following code segment.

ROUTINE FRACCS:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRINAR call, specifying the same control exchange.

Routine Call:

CALL FRACCS (exchange)

with: exchange: Name of the control exchange where the calling task must wait for the protected sequence to become accessible.

Required Stack: 6 bytes

ROUTINE FRRELS:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRINAR call,specifying the same control exchange

Routine Call:

CALL FRRELS (exchange)

with: exchange: Name of the control exchange to which its corresponding message has to be sent by FRRELS.

Required Stack: 2 bytes

ROUTINE FRINAR:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine Call:

CALL FRINAR (exchange)

with: exchange: Name of the control exchange - message combination (15 bytes) which has to be initialized by FRINAR.

Required Stack: 4 bytes

5.2.1.6 System Error Messages

Most of the system and interface routines perform some kind of error checking, particularly when messages are received. In general, the routines branch to an exception code, and call the routine FXSERR if an error is detected. This routine may be supplied by the programmer; a default routine with this name is contained within the I/O library FIORMX.LIB (and hence used by the CGCS). This routine writes the following error message to the console (within the scrolled part of the CRT screen), accompanied by a "beep" signal:

***** SYSTEM ERROR (TASK tsknam, LOC hexl) *****

Within this message, "tsknam" is the actual name of the task, as specified in the configuration module, and "hexl" is the (absolute) hexadecimal address where the call to the routine which detected the system error had been performed. If, for example, a FORTRAN routine calls FRACCS in order to gain protected access to shared resources, and FRACCS detects an erroneous message waiting at the control exchange, the "SYSTEM ERROR" message will contain the name of the task the FORTRAN routine belongs to and the absolute location of the FRACCS call within this routine. The task name and location information provided with the error message does not necessarily mean that the error was caused by this task; most probably some other task is to blame for it. Still, the interface routines lack the ability of clairvoyance, and they can only report an error when it was detected but cannot give any further suggestion what might have caused it. Anyhow, the information contained in the error message may be helpful to detect and remove the error source.
Application routines may also use this facility. Still, a direct call to FXSERR does not necessarily make sense as this routine returns not the location from where it was called but the location from where the routine was called which, in turn, called FXSERR. This was done on purpose in order to give a closer information about the actual point where the error

occurred since the interface routine which performs the actual FXSERR call may have been called repeatedly by the same task. An FXSERR call from a routine which forms the body of a task would even render a completely meaningless "location" value. This can be overcome by calling the routine FXSYSE which performs the required interfacing and returns the location of the FXSYSE call with the error message.

Both routines (FXSERR and FXSYSE) are non-reentrant (i.e., protected by a software interlock) and do not require any parameters.

ROUTINE FXSERR:

Routine Type: Assembly language subroutine; not reentrant; protected by a software interlock.

Initialization: Execution of FXITSK.

Routine Call:

```
CALL FRSERR
```

Required Stack: 8 bytes

ROUTINE FXSYSE:

Routine Type: Assembly language subroutine; not reentrant; protected by a software interlock.

Initialization: Execution of FXITSK.

Routine Call:

```
CALL FXSYSE
```

Required Stack: 10 bytes

System error messages generated by the iRMX-80 control routines:

FXSEND: Too large message length (> 243) was specified by the calling task. No message is sent.

FXWAIT: The type of the received message differs from the type specified with the routine call. The task continues waiting for a correct message.

FXACPT: The type of the received message differs from the type specified with the routine call. The task attempts to receive another message.

FRSEND: none

FRWAIT: See FXWAIT.

FRACPT: See FXACPT.

FRINIT: Too large message length (> 246) was specified by the calling task. The task is suspended.

FRCRSP, FRCXCH, FRDLVL, FRDTSK, FRDXCH, FRELVL, FRRESM, FRSUSP: none

FRACTV: none

FXCFLG: Illegal message detected at the creation/deletion request exchange. The message is ignored.

FXCRFE, FXDLFE: none

FRACCS: Illegal message detected at the access control exchange.

FRRELS, FRINAR: none

5.2.1.7 Free Space Manager Initialization

Prior to being able to request memory from the Free Space Manager, any iRMX-80 application has to supply a sufficient amount of memory to it. This is only possible at execution time; still, it has to be done early enough before memory is requested from the Free Space Manager. The initialization module is generally the most appropriate place to execute this memory transfer, at least the first time.

5.2 System Interface and Auxiliary Routines

In order to permit the initialization of the Free Space Manager from FORTRAN routines, the subroutine FRIFSM was provided. FRIFSM has to be called with the start and end addresses of the memory to be submitted as parameters. Arbitrary memory block lengths may be defined, and FRIFSM may be invoked multiplely. Since the Free Space Manager can only handle memory blocks whose lengths are greater than 8 and multiples of 4, the lengths of the memory blocks submitted (i.e., end address minus start address plus one) should comply with these rules in order to avoid the transfer of unusable memory. FRIFSM ignores all calls with end addresses less than or equal to start addresses; still, it does not check whether at least eight bytes were to be transferred. Note: Memory locations outside the submitted block may inadvertently be changed if blocks shorter than four bytes are submitted!

ROUTINE FRIFSM:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine Call:

```
CALL FRIFSM (start address, end address)
```

with: start address: INTEGER*2 variable or constant holding the address of the first byte to be submitted.
end address: INTEGER*2 variable or constant holding the address of the last byte to be submitted.

Note: (<end address> - <start address> + 1) must be greater than or equal to 8 and should be a multiple of 4. FRIFSM is disabled if <start address> ≤ <end address>.

Required Stack: 2 bytes

5.2.2 Console, Printer, and Buffer Input/Output Routines - Libraries FIORMX.LIB, FIOISS.LIB, FIORXI.LIB, and FIORXR.LIB

NAME	FUNCTION	CHAPTER
FRIOST	initialization routine for I/O funct.	5.2.2.1
FRDATI FRSTRI FRDTBI FRSTBI	data input routine (from console) character string input routine (cons.) data input routine (from user buffer) character string input routine (buffer)	5.2.2.2
FRDATO FRSTRO FRDTPR FRSTPR FRDTBO FRSTBO	data output routine (to console) char. string output routine (to cons.) data output routine (to printer) char. string output routine (to print.) data output routine (to user buffer) char. string output routine (to buffer)	5.2.2.3
FRINMD FROUTM FRPRMD FRINPR FRCLRO FRSPTO FRMCHG	input mode selection routine output mode selection routine (console) printer mode selection routine input prompt string modification CRT screen clearing routine printer timeout setting routine LOGICAL*1 function: output mode changed	5.2.2.4
FRCSTR	control string building routine	5.2.2.5
FRSTHX FRFXIN FXFLIN FRHXOT FRFXOT FXFLOT	conversion ASCII-INTEGER*1 conversion ASCII-INTEGER*2 conversion ASCII-REAL conversion to hexadecimal ASCII string conversion INTEGER-ASCII conversion REAL-ASCII	5.2.2.6

The I/O routines described within this chapter perform input from and output to a console CRT terminal, output to a printer, and input from and output to a user supplied buffer. The latter feature can be utilized to create the output to a text type disk file, or to read such a file, respectively.

In order to permit a reasonable overhead for the application routines (particularly, a reasonable stacksize), the following program structure was chosen: Rather small reentrant modules are called by the user task, using its stack, in order to

build I/O request messages. These messages are sent to the entry exchanges either of the input task INDATX or of the output task OUTDTX. These tasks perform the required conversions and request in turn input from the iRMX-80 Terminal Handler, or they send output to it, if applicable.

Despite of the reentrancy of the interface routines, there is a considerable time delay inherent with each I/O operation as each interface routine has to wait for a response of the I/O task it called. First, the message requesting an I/O operation has to queue at the entry exchange of the corresponding input or output task. Second, the conversion routines themselves may require a considerable execution time, and third, the I/O task has to wait for the response of the Terminal Handler. This response is - in the case of an output request - delayed by the time required for sending the output string to the console or printer. Input requests may even be detained for an indefinite time until the operator entered an input line. (Still, this does not mean that the processor is totally busy with the I/O action and cannot execute any other task meanwhile.) The execution of INDATX can only be resumed after a complete logical input line was entered on the console. It is therefore very likely that one task is most time waiting for a command entry issued by the operator. INDATX is then actually in a permanent wait for a response of the Terminal Handler. Furthermore, the echo output - which may be generated after the input of a line from the Terminal Handler - requires the availability of the output task. If already several other tasks are queued at the entry exchange of OUTDTX, the release message of INDATX will only be issued after all these output requests were processed. Even operations which do not require the service of the Terminal Handler, i.e., I/O from/to a user supplied buffer, suffer from these delays since their pertinent output requests have also to queue at the input exchanges of INDATX or OUTDTX.

Therefore, a task with a critical timing (or, which is equivalent in most cases, an interrupt service task) should never perform any input or output operation. This applies even to apparently low-speed tasks: a timer task which runs only once a second, performing a timed wait (for 20 RMX time units at 50 ms each) meanwhile, will become inaccurate if it includes an output operation. This is true because the timed wait will delay the task in any case until 20 clock ticks have passed. If this task, however, has to wait somewhere else, for example for the response of OUTDTX, some clock ticks may or may not already have happened while the task was <u>not</u> waiting at the timing exchange. (The auxiliary timer task FXTIME has, therefore, to use a special approach for writing its time information to the console screen.)

Any task requesting output has to wait until the output action was performed. Otherwise, memory locations could be changed before or - even worse - while they are being processed by the output routines. This imposes also the demand that the memory locations which were specified by the task requesting output must not be changed by any other task. (The necessity for an input requesting task to wait until the input was done is evident.) Therefore, it is advantageous to provide dedicated tasks which perform output operations but no operations which might be urgently needed somewhere else. Splitting the generation of output between several tasks might be a good idea if the system is rather complex, particularly, if large amounts of output are periodically generated. In this case, several tasks can improve the system's performance since they can queue at the entry exchange of OUTDTX, thus providing some kind of buffering. In contrast, console terminal input should be requested by one task only (compare chapter 3.1.6). This restriction does not apply to I/O from/to a user supplied buffer: The task requesting buffer I/O submits a buffer of its own with the I/O request, and this buffer is handled whenever the I/O tasks find time to do so. The sequence of buffer I/O requests is therefore irrelevant, provided they apply to different buffers for each task.

INDATX uses two input sources, namely the console CRT and a user supplied buffer, and OUTDTX can output to three channels, namely, the console CRT, the printer, and a user supplied buffer. Device I/O is done by the Alternative Terminal Handler (compare chapter 3.3.4) which supports console CRT I/O and, in addition, the output to a printer performed via an additional USART on an I/O expansion board. Only the Alternative Terminal Handler must be used in conjunction with the modules in FIORMX.LIB; only this Terminal Handler supports two I/O devices and directly addressed CRT terminal output.

The I/O tasks do not receive messages directly from the routines requesting I/O operations. Two small reentrant subroutines are provided for input and output, respectively, which build messages and wait for the response of the I/O tasks. This was done in order to off-load the application software from the overhead of creating a message for each I/O operation. Each interface routine has several entry points; some of them trigger directly an input or output operation, some can be used to set parameters of the I/O routines. Separate entry points are provided for the input and the output of CHARACTER variables and of variables of any other type. This was necessary because FORTRAN uses different parameter passing conventions for these two groups. While a normal variable - no matter what its type is - is passed by its address only, FORTRAN passes two parameters for each

CHARACTER variable, namely, its (start) address and its length. (The length of a CHARACTER variable is determined when the FORTRAN program is compiled; it is either set by the declaration or by the length of a string contained in the source code.) It is essential that the appropriate interface routine is invoked for a certain parameter. Calling a string I/O function with a non-CHARACTER parameter or vice versa will inevitably cause a disastrous system crash. There is no possibility whatsoever for the interface routines or the I/O tasks to determine the actual type of a parameter.

The scheduling of output to the console CRT (which is supposed to be permanently connected to the system) and to a user supplied buffer is straightforward: A request message is sent by the interface routine to the entry exchange of OUTDTX, and the calling task waits within the interface routine for the return of the request message which happens when OUTDTX has completed its work. OUTDTX, in turn, sends an output request message of its own to the Output Terminal Handler (if applicable), and waits until this request message is returned upon completion. Only upon receipt of the returned request message, the task requesting output is released from its message-exchange interlock. This approach was no more suitable for the scheduling of printer output, since the printer may be disconnected or unable to receive data for a prolonged period while it empties its buffer. OUTDTX would be detained for an indefinite time in this case if it would wait for the Terminal Handler's response after the attempted output to the printer. Console I/O would thus also be delayed unnecessarily. Therefore, a different scheduling approach was chosen for the printer output: The output requesting task waits (within the interface routine) first for the printer output request message of OUTDTX which is returned to this access control exchange either by OUTDTX or by the Terminal Handler, depending on whether information was only added to the printer buffer in OUTDTX, or whether the buffer was actually dispatched for printing. Only when this message was received, the interface task sends the request message to the entry exchange of OUTDTX. Tasks requesting output from a (not ready) printer queue therefore at the access control exchange, and they are released only either when the printer is operable again, or after a time-out (in which case an error message is displayed in the scrolled portion of the console CRT). OUTDTX is, however, not affected by a not-ready printer.

```
TASK NAME:        INDATX
ENTRY POINT:      FXINTI
STACK LENGTH:     184 (a) or 174 (b) bytes
PRIORITY:         134 (higher than all routines which
                  request input)
DEFAULT EXCH.:    none
EXTRA:            18 (a) or 13 (b) (see chapter 3.1.5.1)

(a) ... for software floating-point arithmetics
(b) ... for hardware floating-point arithmetics

INITIAL EXCH:     FXINDT


TASK NAME:        OUTDTX
ENTRY POINT:      FXINTO
STACK LENGTH:     200 (a) or 181 (b) bytes
PRIORITY:         135 (higher than all routines which
                  request output)
DEFAULT EXCH.:    none
EXTRA:            18 (a) or 13 (b) (see chapter 3.1.5.1)

(a) ... for software floating-point arithmetics
(b) ... for hardware floating-point arithmetics

INITIAL EXCH.:    FXOUTD
                  FXPRAC
```

The above mentioned stack length values were calculated from the stack length information included with the FORTRAN floating-point routines invoked by INDATX and OUTDTX. They are necessarily a worst-case estimation which is never fulfilled for an actual execution since the stack value is calculated by the ISIS-II Linker as the sum of the stack requirements of all routines which will never all be active at the same time. During practical use, a stack of 150 bytes was found to be by far sufficient for INDATX as well as for OUTDTX.

5.2.2.1 Input/Output Initialization

Two different types of initialization have to be distinguished in this particular context: first, the I/O tasks INDATX and OUTDTX themselves have to be initialized, which, however, need not be of further concern here. Explicit initialization is required, though, for the interface between application tasks and the I/O modules.

Special precautions have to be taken when the Static Task Descriptors of the tasks are defined. Each task which uses FORTRAN floating-point arithmetics has to add either 13 or 18 additional bytes to its Task Descriptor, depending on whether hardware or software floating-point arithmetics was chosen (compare chapter 3.1.5.1). In addition, each task that uses the I/O routines has to add two more bytes to its Task Descriptor in order to permit the installation of a pointer to the exchange-message combination required by the I/O routines. Only one value for such an offset is legitimate within the entire system. If none of the tasks which perform I/O uses FORTRAN floating-point functions, an "EXTRA" value of 2 may be chosen. Otherwise, the "EXTRA" value of <u>all</u> tasks which use the routines in FIORMX.LIB must be set to 15 (for a system including an iSBC 310 High Speed Mathematics board) or to 20 (if software or 8231-based floating-point arithmetics is used) if one or more tasks perform not only I/O but also floating-point operations. A corresponding offset value - 0, 13, or 18 - has also to be specified at system linkage time, together with some other system constants (compare chapter 5.2.2.8).

Furthermore, each task which will perform I/O operations has to call the initialization routine FRIOST once and only once. The task has to specify the name of a 31 byte location in read-write memory where FRIOST can build an I/O request message and the response exchange for the I/O tasks. It is this address to which the pointer added to the task's Task Descriptor points. A task can only perform either input or output at a given time; therefore, only one I/O request message is required for a task. Linking this message (and the response exchange) to the Task Descriptor of the task allows to omit this information in any future I/O request as the interface routines can independently determine these addresses. This saves a considerable program code overhead.

ROUTINE FRIOST:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine Call:

```
CALL FRIOST (exchange)
```

with: exchange: Name of a 31 byte long memory location where FRIOST can build an I/O request message and a response exchange for the I/O tasks.

Required Stack: 2 bytes

5.2.2.2 Input Routines

5.2.2.2.1 Programming Interface

Two console input routine entry points are provided, FRDATI and FRSTRI. FRDATI can handle all kinds of variables - INTEGER (including LOGICAL), REAL, and Hollerith type - except variables of the type CHARACTER which must be input by FRSTRI. Two routines, FRDTBI and FRSTBI, are provided for "input" (i.e., conversion from ASCII into binary numeric notation) from a user-supplied buffer. The contents of the user supplied buffer are scanned for the requested input in this case, rather than data obtained from the Terminal Handler. No echo output is generated on the console. Therefore, multiple tasks may concurrently use the buffer input feature, provided they submit different buffers.

The input routines must be called as follows:

```
CALL FRDATI (control string,variable,status)
    or
CALL FRSTRI (flag string,character variable,status)
    or
CALL FRDTBI (control string,variable,buffer,status)
    or
CALL FRSTBI (flag string,character var.,buffer,status)
```

<control string>:

<control string> can be either a string, enclosed between single quotes, or a CHARACTER type variable, holding the following information:

control string := 'flag,type'

where <flag> is an integer which determines the actual input operation, and <type> is a single character (upper- or lowercase) which controls the kind of conversion performed by the input routine.

<flag string>:

No <type> identification is required in the case of FRSTRI or FRDTBI; the control string comprises therefore only the value of <flag>, enclosed in single quotes, or the name of a CHARACTER variable holding a corresponding string.

<variable>:

<variable> is the name of a single variable whose contents are to be set by INDATX. The type of this variable should correspond to the <type> specified with the control string in order to avoid the destruction of other data if INDATX reads more bytes than are reserved for the specified variable. Only CHARACTER variables may be used in a FRSTRI call. The length of a string which is input via a FRSTRI call is automatically determined by the dimension of the CHARACTER parameter. (A FRSTRI call with a CHARACTER*10 variable specified as a parameter will, for example, read 10 characters.)

Note: INDATX always returns a signed two-byte integer (INTEGER*2) if "I" was specified as <type>. It is, therefore, not possible to assign an input value directly to an INTEGER*1 variable.

<status>:

After each input request, INDATX returns a one byte Boolean variable to the parameter <status> which indicates the result of the input request. The value of this variable is .FALSE. (00H) if the input request could be fulfilled; it is .TRUE. (0FFH) if either an input error occurred or if the input string was empty. No new value is assigned to <variable> if <status> was set to .TRUE.; the application program should check this flag in order to determine whether it did receive new data or not. It is recommended to use a LOGICAL*1 variable as <status> parameter; this variable can easily be checked in a logical IF statement.

<buffer>:

This parameter should be the first byte of a user supplied buffer. The buffer should be declared as an INTEGER*1 array under FORTRAN; it must by no means be a CHARACTER type variable. The first byte of the buffer (i.e., the first element of the array) must be set to the length of the actual buffer (which starts at the third element of the array) prior to the call to FRDTBI or FRSTBI. The second byte is reserved for INDATX use and should not be modified by the user; the actual contents of the buffer start at byte 3. The buffer may be referred to either by BUFFER or by BUFFER (1) within an FRDTBI or FRSTBI call; prior to the call, the data must be provided in the buffer beginning with BUFFER (3) which is to be converted according to <type> and to be written to <variable>. INDATX may

parse up to the number of bytes within the buffer which was specified by BUFFER (1).

Example:

```
        INTEGER*1 BUFFER (130)
C           (The buffer proper should contain 128
C           bytes in our example, plus the two
C           length bytes)
        BUFFER (1) = BUFLEN
C           (BUFLEN must be <= 128)
        DO 100 I = 1,BUFLEN
        BUFFER (I+2) = ...
C           (Place data into the buffer)
100     CONTINUE
        CALL FRDTBI ('1,E',X,BUFFER,STAT)
C           (Scan the buffer for a floating-point
C           number, convert it and store it in X)
```

The input functions and conversion algorithms are selected by means of the parameters <flag> and <type>. <flag> affects the handling of the input line which was entered by the operator or contained in the user-supplied buffer, and <type> controls the conversion algorithms:

* <flag>:

<flag> controls eight functions of INDATX which are mapped to the eight bits which represent the one-byte integer <flag>. Its value can be calculated as follows:

```
<flag> = 0   (if the current input buffer contents are to be
             processed, starting at the input pointer, i.e.,
             after the character last read)
     +   1   (if a new input line is requested; in the case of
             buffer input, the buffer pointer is reset to the
             beginning of the input buffer)
     +   2   (if input is to be performed after the input
             pointer was reset to the beginning of the line
             which is currently kept in the buffer)
     +   4   (if input is to be performed after the input
             pointer was moved to the next blank)
     +   8   (if input is to be performed after the input
             pointer was moved to the next digit)
     +  16   (if input is to be performed after the input
             pointer was moved to the next character)
     +  32   (if the echo output is to be suppressed)
```

+ 64 (if input is to be performed after the input pointer was advanced to the first floating-point number)
+ 128 (if input is to be performed after the input pointer was advanced to the first non-blank character)

The interdependence between the bits 0 through 4 and the corresponding functions is shown in the following table in order to make the use of the <flag> parameter easier. Bit 5 which suppresses the echo output is not included in the table in order to enhance its clarity; a value of 32 has to be added to the appropriate <flag> value shown below if it is to be set. The highest two bits, 6 and 7, are hardly to be used in conjunction with other values than 0, 1, or 2.

	ADVANCE TO NEXT ALPH.CH. | ADVANCE TO NEXT DIGIT | ADVANCE TO NEXT BLANK | RESCAN INPUT LINE | READ A NEW LINE			ADVANCE TO NEXT ALPH.CH. | ADVANCE TO NEXT DIGIT | ADVANCE TO NEXT BLANK | RESCAN INPUT LINE | READ A NEW LINE	
<flag>			<flag>		
0	0 0000		16	1 0000	
1	0 0001		17	1 0001	
2	0 0010		18	1 0010	
3	0 0011	*)	19	1 0011	*)
4	0 0100		20	1 0100	
5	0 0101		21	1 0101	
6	0 0110		22	1 0110	
7	0 0111	*)	23	1 0111	*)
8	0 1000		24	1 1000	
9	0 1001		25	1 1001	
10	0 1010		26	1 1010	
11	0 1011	*)	27	1 1011	*)
12	0 1100		28	1 1100	
13	0 1101		29	1 1101	
14	0 1110		30	1 1110	
15	0 1111	*)	31	1 1111	*)

*) The rescan option is inoperative if a new input line was requested.

Setting the lowest-order bit of <flag> (bit 0) in a FRDATI or FRSTRI call makes INDATX request a new input line from the Terminal Handler prior to performing any conversion. The previous contents of the input buffer are overwritten, and data are lost which have not yet been read from the previous

input line. In buffer input mode, the function of bit 0 is identical to that of bit 1. Either one of those bits should be set when new user supplied buffer contents are submitted for scanning.

The second bit (bit 1) determines whether the pointer within the line buffer is to be reset to the beginning of the buffer prior to reading. An input request with this bit of <flag> set can process the entire input line, from its beginning, while otherwise the requested data are taken from the characters in the input string indicated by and following the current input pointer position. The input pointer is normally set to the first character that has not yet been processed, and the next input request is satisfied starting at this position. The rescan option moves the pointer back to the beginning of the line, thus permitting the same input line to be processed by repeated read commands.

The third, fourth and fifth bits (bit 2-4) allow data selection within the input line. They make the input pointer advance to the next blank, digit, and alphabetic character, respectively. ("Blank" means spaces and control characters such as "tabs", as far as they are not regarded as commands by the Terminal Handler and therefore stripped from the input line. "Digits" refers to the characters "0" through "9", and "alphabetic", to "A" through "Z", upper- or lowercase.) This feature is very helpful if combined entries - e.g., entries consisting of command keywords and numbers - have to be processed. Setting more than one of these bits makes INDATX first search for the first blank following the current position of the pointer, then for the first digit, and finally for the first alphabetic character (if applicable). Note that signs are lost during numeric input if bit 4 is set. In order to scan for signed numbers, or numbers in floating-point notation which may start with a decimal point, it is advisable to use bit 6 rather than bit 4 (see below).

The sixth bit (bit 5) permits, if set, to suppress the generation of echo output in the split-screen mode. This can be helpful if information is to be added to an input line before it is echoed back. This flag bit is ignored in the completely scrolled screen mode where the echo of the input line remains on the screen anyhow, and it is ineffective unless a new input line was requested (bit 0 set). Since no echo output is generated by the buffer input routines, bit 5 is ineffective in this case, too.

Bit 6 permits the search for numbers in all permitted fixed- and floating-point notations. The pointer is advanced until any one of the following characters is detected: "0" ... "9",

"E" or "e", ".", "+", and "-". Numeric entries which would be mutilated if bit 4 were used can thus be processed properly.

Bit 7, finally, advances the pointer to the first non-blank character. This switch permits to skip blank portions of the input line.

* <type>

The <type> parameter can be one of the following characters or strings:

Aw: The next w (w < 128) characters, beginning with the position of the pointer, are copied into memory locations beginning with <variable>. The programmer has to make sure that the specified string length w does not exceed the available memory area which is reserved for <variable>. (In most cases, <variable> will be the name of an array. It must by no means be the name of a variable declared CHARACTER although it is possible to assign data to a CHARACTER variable by reading it into an array of, say, type INTEGER*1 and the same length, which is linked to the CHARACTER variable with an EQUIVALENCE statement. Still, the FRSTRI call is more convenient for reading CHARACTER variables.) The input string whose length was specified with w is filled in any case: if the logical end of the input string (a carriage return) is encountered before w characters were read, the remainder is filled with spaces. The string may even exceed the length of the input buffer. In this case, a new input line is requested from the Terminal Handler, and the new input is added "seamlessly".

B: The input string is interpreted as a hexadecimal number. Input is terminated if any character except "0" ... "9", "A" ... "F", or "a"... "f" is detected. An arbitrary number of hex digits may be entered; only the value corresponding to the last two digits is stored in the (INTEGER*1) variable supplied by the calling routine.

E,F: The next contiguous string of numeric characters is interpreted as and converted to a floating-point number. "E" and "F" are equivalent commands. The numeric string is considered terminated when a character other than a sign (accepted only in the first positions of the string or of an exponent), a decimal point, an "E" (upper- or lowercase, accepted only once

within a number), or any of the numbers "0" through "9" is encountered. The numeric string must not extend beyond the physical end of the input buffer. If the end of the buffer is encountered before the numeric string was terminated, an error message is output, and no value is assigned to the REAL variable passed as a parameter by the requesting task. No formatting restrictions apply; numbers can be entered in any arbitrary combination of floating-point or scientific notation (compare the examples in chapter 5.2.2.2.2). Leading spaces are ignored. The full range of REAL variable values is supported. Numbers greater than the maximum floating-point number (±3.4E38) are entered as the greatest permissible number; underflow numbers (less than ±1.17E-38) are set to zero.

I: The next contiguous string of numerals is converted into an INTEGER*2 variable (two bytes, signed). The end of the string is recognized if another character than "0" through "9" is encountered. The string must not extend beyond the physical end of the input buffer. Aside from this demand, any input string length and therefore any magnitude of the input number is permissible; still, the numbers will be treated modulo (32768) (compare chapter 5.2.2.2.2).

W: The input string is interpreted as a hexadecimal number. Input is terminated if any character other than "0" ... "9","A" ... "F", or "a" ... "f" is detected. An arbitrary number of hex digits may be entered; only the value corresponding to the last four digits is stored in the (INTEGER*2) variable supplied by the calling routine.

Xw: The next w (w < 128) characters are skipped, beginning with the current position of the input pointer. A new input line is requested if the length to be skipped extends beyond the physical (but not the logical) end of the input line.

Zw: A total of w bytes, beginning with the location specified by <variable>, is set according to the up to (2*w) non-blank characters following the input pointer position (leading spaces are skipped, though). These characters are interpreted as hexadecimal digits. The left-most digit is stored in the highest-order location, and so on. The transfer of the data is terminated either if the specified number of bytes was filled or if any character but "0" through "9" and "A" through "F" (upper- or lowercase) was encountered.

The remaining half-bytes which could not be set by the input string are set to zero in this case. (Note that the input pointer remains at the location where the first non-hex character was found; the next read request will start at this position.) A continuation of the input string across the physical input buffer boundary is possible.

Note: An omitted numeric extension of the "A", "X", and "Z" type commands is interpreted as zero.

Sample calls:

```
LOGICAL*1 STAT
...
CALL FRDATI ('1,F', X, STAT)
IF (STAT) ... (error exception routine)
```

This call assigns a new floating-point value to the REAL variable X. The input buffer is cleared, and a new input line is requested prior to converting the decimal string into the binary floating-point representation.

```
CHARACTER*10 CHAR
...
CALL FRSTRI ('0', CHAR, STAT)
```

The next ten characters, beginning with the current position of the input pointer, are transferred to the CHARACTER variable CHAR. The input buffer has to contain the required information from an earlier input request. The remainder of the variable CHAR is filled with spaces if the logical end of the input line (a carriage return) is encountered. No input is requested from the operator.

```
CHARACTER*3 CONTRL
CONTRL = '1,F'
...
CALL FRDATI (CONTRL, X, STAT)
```

This call is identical to the first example. Using a CHARACTER variable as <control string> parameter may save some typing, particularly if the same control string is repeatedly required. (It has no effect, however, on the program code length as identical strings are allocated to the same internal location by the FORT80 compiler.)

INDATX has a built-in error detection facility: Erroneous or missing parameters within the control string or fixed- or floating-point numbers which extend beyond the physical end of

the input buffer are reported on the console CRT as an "INPUT ERROR". In this case, the <status> flag is also set to .TRUE., and no new value is assigned to the <variable> location.

ROUTINE FRDATI:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Required Stack: 14 bytes

ROUTINE FRSTRI:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Required Stack: 14 bytes

ROUTINE FRDTBI:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Required Stack: 14 bytes

ROUTINE FRSTBI:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Required Stack: 14 bytes

5.2.2.2.2 Operator Interface

The input routines are designed such that a maximum of data safety can be combined with a maximum of convenience for the operator. The following table shows the resulting internal data for different input strings, read with different <type> parameters.

INPUT STRING	A4	E,F	I	Z4
ABCDEFGHIJK	"ABCD"	0.	0	ABCDEF00
#)	"EFGH"	1.0000	0	EF000000
XYZ	"XYZ "	0.	0	00000000
123.456	"123."	123.456	123	12300000
1.5E-3	"1.5E"	0.0015	1	10000000
+4321.012E-002	"+432"	43.21012	4321	00000000
-E4	"-E4 "	-10000.0000	0	00000000
E99	"E99 "	3.40E+38	0	E9900000
32767	"3276"	32767.0000	32767	32767000
32768	"3276"	32768.0000	0	32768000
32769	"3276"	32769.0000	1	32769000
-32767	"-327"	-32767.0000	-32767	00000000
-32768	"-327"	-32768.0000	-32768	00000000
-32769	"-327"	-32769.0000	-1	00000000
40490FDB *)	"4049"	40490.0000	7723	40490FDB

#) If <flag> bit 6 (compare chapter 5.2.2.2.1) is set.

*) This is the floating-point representation of PI (3.14159264).

(Note that the hexadecimal representation in the right-most column was adjusted to show the highest-order byte at the left. The internal storage of floating-point numbers reverses this orientation, i.e., the lowest-order byte is assigned to the lowest address, and the highest-order byte, to the highest.)

Input rules for fixed- and floating-point numbers:

* Leading spaces or zeros are ignored.

* The number is terminated by the first character which is not permitted for the particular format or by the logical end of the input line (carriage return).

* If several numbers are requested within one input entry, they have to be separated by spaces (unless other demands are imposed by the application software). Any number of separating spaces is permitted. It is, however, possible to use arbitrary delimiters between multiple input numbers if the <flag> bit 6 (advance to next number) is set.

* Plus signs are automatically assumed if no "minus" is entered.

* Floating-point numbers may be entered as a (signed or unsigned) exponent only, omitting the mantissa. The character "E" or "e" is therefore interpreted as 1 (1.E0).

* A decimal point is assumed after the last digit of a floating-point number if no decimal point was entered.

Input Line Editing and Control Characters:

The I/O routines support the line editing features and the control character set of the Alternative Terminal Handler. For a full explanation of the control codes processed by the Alternative Terminal Handler refer to chapter 3.3.4.2 or Appendix 5. The following summary shows the line editing and control commands valid for the FIORMX I/O routines. Note that ESC (Escape) is used as a line deletion rather than a line termination command by the I/O Interface routines.

All control characters except the following ones are rejected by the Terminal Handler (except when entered immediately after a "Cntl-P"):

CR (Carriage Return) - Line termination.
LF (Line Feed) - Line Termination.
RUBOUT - Delete last character.
ESC (Escape) - Delete current input line.
Control-X - Delete entire input buffer.
Control-Z - Delete current input line, and return end-of-file (compare chapter 5.2.2.7).
Control-R - Re-write the input line on the console screen.
Control-P - Accept control characters literally.
Control-S - Halt output to the console.
Control-Q - Resume output to the console.
Control-O - Discard or resume output to the console.
Control-E - Halt output to the printer.
Control-F - Resume output to the printer.
Control-V - Discard or resume output to the printer.
Control-C - Invoke iRMX-80 Debugger (not used in the CGCS).
Control-A - Exit from iRMX-80 Debugger (not used in the CGCS).

"Cntl-C" and "Cntl-A" can be locked out if the Alternative Terminal Handler flag location RQDBEN (debug enable) is reset to a zero value (which is done in the CGCS).

5.2.2.3 Output Routines

Six routines are supplied within FIORMX.LIB which permit the output of CHARACTER and of other type variables to the console, to the printer, and to a user supplied buffer. FRDATO, FRDTPR, and FRDTBO permit the output of all variables (except of type CHARACTER) to the console, to the printer, and to a user-supplied buffer, respectively, whereas FRSTRO, FRSTPR, and FRSTBO are exclusively provided for handling CHARACTER variables. While the console and printer output routines generate actual output, the buffer output routines only deposit a string which is obtained by the conversion of (binary) data within a user-supplied buffer. The further handling of this buffer is under the responsibility of the application routine.

The console output routines can be programmed to generate either a normally scrolled or a split screen output. In the normally scrolled mode, output (and the input echo) is added after the last line written to the screen, i.e., most of the time in the last line of the screen. The previous output moves up by one line.

In contrast, the output area is divided into three main zones if the split-screen mode was selected: Part of the screen can be directly addressed, and there is no relation between the position of an output string on the CRT screen and the time when it was written. Another part of the screen forms a scrolled area. Output written to this area is added in its last line, and the previously written lines move up by one line. The software overhead for generating such a scrolled output is, however, relatively large as each line of the scrolled block has to be re-written each time a new line is added. For a scrolled block of eight lines, therefore, eight lines have to be written when a one line output is requested. (This is inevitable as this type of scroll is no more performed by the hardware of the terminal but by the software of OUTDTX.) With regard to the large overhead, the size of the scrolled block and the output to it should be confined to a minimum consistent with the application. (Hence, the CGCS uses a scrolled area of five lines rather than the default of eight lines.) Any echo output, if requested, is also directed to the scrolled block. The third area on the screen, finally, is the input area which may but need not be contiguous to the scrolled block. While the scrolled block can be freely moved on the screen under execution-time control, the position of the input area is fixed at the bottom of the screen. Its size must be defined as a configuration constant; it must be large enough to permit the cursor to perform the jump to the next line without leaving the screen area when a carriage return is

entered. Otherwise, a (hardware) scroll would ensue and, inevitably, a lot of confusion in the next data output. The Alternative Terminal Handler supports a default input line length of 80 characters. An input area of two lines is therefore only sufficient if no input prompt string (compare chapter 5.2.2.4.4) is to be used (which applies to the CGCS). Otherwise, the number of input lines kept in the constant F0LINC (compare chapter 5.2.2.8) should be set to 3. Note, however, that the Alternative Terminal Handler clears two lines in the input area (the input line proper and the line following it); therefore, no information can be reasonably written to the bottom line on the screen if F0LINC was set to 2. The last line of the screen is generally only required as a buffer zone for the cursor; it is not automatically cleared if F0LINC is set to a value greater than 2. It can therefore be freely used for directly addressed output. (Note that directly addressed output can be written virtually anywhere on the screen. It will, however, be eventually overwritten or deleted if directed to the scrolled area or to the input area.)

The output routines must be called as follows:

```
CALL FRDATO (control string,variable)
    or
CALL FRSTRO (control string,variable)
    or
CALL FRDTPR (control string,variable)
    or
CALL FRSTPR (control string,variable)
    or
CALL FRDTBO (control string,variable,buffer)
    or
CALL FRSTBO (control string,variable,buffer)
```

<control string>:

<control string> can either be a string, enclosed in single quotes, or a CHARACTER variable which holds the appropriate information (compare chapter 5.2.2.5). The control string must have the following format:

a) for FRDATO, FRDTPR, and FRDTBO:

```
control string := 'line,column,format'
```

b) for FRSTRO, FRSTPR, and FRSTBO:

```
control string := 'line,column'
```

A <format> may be specified also for FRSTRO, FRSTPR, and FRSTBO; it is, however, ineffective.

The reason why a control string was chosen rather than separate parameters is that this approach saves considerably FORTRAN code, compared to the specification of different parameters. One of these, <format>, would have been a CHARACTER type variable anyhow. The control string approach adds also safety to the system as an omitted parameter within the control string can under no circumstances affect the system operation. Conformity reasons demanded the extension of this approach to the input routines where its advantages are less stringent.

<variable>:

<variable> can be the name of a variable whose type corresponds to the output routine and to the <format> parameter, if applicable. It may also be a constant. Note that string output is permissible only with FRSTRO, FRSTPR, or FRSTBO; a suitable string may be specified directly as a subroutine parameter.

<buffer>:

<buffer> must be the name of a suitable buffer array (by no means of type CHARACTER!). Usually, an INTEGER*1 array will be chosen. The first two bytes of this buffer area (i.e., the first two elements of an INTEGER*1 array) are used as control bytes: The first byte has to hold the available number of bytes within the buffer region proper, stored there prior to the output routine call; the second byte is used by OUTDTX and should not be changed by the application program. It holds the number of the location within the buffer proper which follows the last location written to. The buffer proper starts with the third byte; OUTDTX builds an output line there. Note: Although the structure of the buffers and control bytes is identical for INDATX and OUTDTX, the same buffer should be used for input and output with great care only. In particular, the buffer pointers which are kept in the second byte of the buffer are changed by both routines. (This does not matter, however, if the input buffer pointer is set to the start of the buffer (with <flag> bit 0 or 1) before each input action, and if the output pointer is not used because a non-zero <column> value is passed with each output call.)

The output routines permit random positioning of the output strings on the console, on the printer, and in the output buf-

fer. Within the scrolled output area on the console, on the printer, and in the buffer, positioning is only possible within the current output line. In the non-scrolled area of the console CRT terminal, every position on the screen can be arbitrarily accessed. These features uncouple the position where output is displayed from the order in which it was generated.

The parameters <line> and <column> permit the random addressing of output screen or line locations. <format> controls the conversion which has to be performed by OUTDTX. The <line> and <column> parameters and the numeric extensions of <format> must be unsigned integer numbers. Any value is permitted (as far as there are no limits imposed by the corresponding function); the submitted value is treated modulo(128). Leading spaces or zeros are ignored. The parameters must be separated by commas. Using a string as a parameter of the output routine calls implies that the location of an output on the CRT screen is basically fixed. A dynamic addressing which permits the calculation of the output location by the application routine can be accomplished with the CHARACTER*16 function FRCSTR with appropriate parameters as <control string> (compare chapter 5.2.2.5).

* <line>

The meaning of this parameter depends on the output device chosen:

(a) CRT output in split-screen mode:

<line> indicates the line in which the output generated by OUTDTX is to be placed. Any value between 1 and the maximum number of lines on the console CRT screen (which must be specified at program configuration time) is interpreted as a line number. Two (ranges of) values of <line> have a special meaning to OUTDTX: a zero value indicates that the output generated by OUTDTX is to be placed into the current output line of the scrolled area on the screen, without sending this line to the console. This permits building an output line by repeated output requests with <line> equal to zero. Any value of <line> greater than the number of lines on the CRT screen puts the output into the current output line of the scroll buffer, too, but transfers this line immediately to the console.

(b) CRT output in completely scrolled mode:

In completely scrolled mode, three output functions can be selected with the <line> parameter:

<line> = 0: Output is built in the output buffer but the buffer is not yet transmitted to the Terminal Handler.

<line> = 1: The contents of the output buffer (including the items added by the output call with <line> set to 1) are output on the CRT console. No Carriage Return - Line Feed pair is appended. This switch permits, for example, to write an input prompt to the console which is to be continued by user-supplied input.

<line> > 1: A CR-LF pair is appended to the line built in the buffer, and the line is output. The output buffer of OUTDTX is cleared after each output operation.

(c) Printer output:

For printer output, any <line> value other than zero will make OUTDTX print the printer output buffer after having added the output item passed with this call. A zero <line> value permits - similar to the console routines - the collection of data in the printer buffer without printing the line. There is no option comparable to <line> = 1 for the printer, though. Note that the scroll buffer and the printer buffer are automatically cleared after having been dumped to the output device.

(d) Buffer output:

A zero <line> value makes OUTDTX perform the proper conversion, and deposit the ASCII string obtained from the conversion routines within the buffer. A non-zero <line> value makes, in addition, OUTDTX append a CR-LF pair immediately after the last item output. (Note that this might lead to problems if an output line is not built in the user-supplied buffer in a conventional "from left to right" mode but by random positioning of various output items. In this case, a dummy output can be made to a position after the last output item in order to place the CR-LF pair correctly.)

* <column>

Similar to <line>, any <column> value between 1 and the maximum number of characters on a CRT screen line, on the printer, and in the user-supplied buffer, respectively, is interpreted

as the start position of the output string requested. (The CRT and printer line lengths have to be specified within a configuration module; the usable user-supplied buffer length must be placed into the first byte of the buffer array.) This applies in any output mode. For a zero column value, the new output is located beginning with the column immediately following the (temporally) last column to which output was written in the scrolled area. The use of a zero column value does not make sense for directly addressed output, it may, still, prove helpful if a scrolled, printer, or buffer output line has to be built. However, absolute and relative addressing should not be mixed under any circumstances. An output request with a zero column value, following absolutely addressed output to locations at the left end of the line, might overwrite other output located to the right of its start column. Using the relative addressing (with the zero column value) is recommended if independent blocks have to be closely packed within a line and if their chronological order should be maintained. On the other hand, tables should be generated with absolute addressing.

Special actions are taken if either the start column value exceeds the permitted line length, or if the output string starting at this position would exceed it. In the first, for buffer output also in the second case, an "OUTPUT ERROR" message is sent to the scrolled block, and the output request is ignored. The treatment of the second case depends on whether scrolled or unscrolled output was requested (printer output is considered as scrolled): In the case of scrolled output, the contents of the line buffer are sent to the output device, the buffer is cleared, and the output string is moved to the left end of the next output line. No more printer output can be written to this second line since it is immediately sent to the Terminal Handler for output. Otherwise, output processing is terminated with an "OUTPUT ERROR" message.

<format>:

The interpretation of the bytes beginning with the location specified with <variable> depends on the contents of the <format> string. Generally, this string consists of one character (upper- or lowercase) followed by one or two integer numbers which must be separated from each other by a period. Leading or trailing spaces are permitted.

REAL variables are rounded to the specified number of digits; although any number of digits may be requested, accuracy is limited to slightly more than seven places. In order to save

execution time, only nine digits are actually converted; if more were specified, the least significant digits are set to zero. The following <format> commands are permitted:

Aw: The w (w < 128) bytes following the location specified with <variable> are interpreted as ASCII characters and therefore converted to a string with length w. (Note that non-printable characters are internally counted like printable characters, which might cause confusion if two strings are to be fitted together in separate output requests.)

Ew.d: A (four byte) REAL variable is converted into the scientific notation format. The total string length reserved for this variable is w; d indicates the length of its fractional part. In any case, w must be greater than (d + 6); otherwise, an "OUTPUT ERROR" message is generated, and the output request is skipped. The parameter d may be any positive integer, including zero. The output is right justified in its reserved area; its general form is sd.ddddEsdd where s is a sign ("+" is only output in front of the exponent), and d are digits. Positive and negative overflow, and indefinite values are indicated by a "+", a "-", and a "?", respectively, preceding a string of asterisks in the mantissa area.

Fw.d: A REAL variable is converted to a floating-point representation. The output string length reserved is w, the length of the fractional part, d. An "OUTPUT ERROR" message is issued if w is less than (d + 3). The output is right justified within its reserved area. Floating-point error conditions are reported as above; in addition, a format overflow is indicated by a string of asterisks instead of the digits of the number (see example).

Gw: This format is a hybrid between the "E" and the "F" formats: a REAL variable is converted into a representation which requires exactly w characters (w must be greater than 2 lest an "OUTPUT ERROR" is reported). The variable is converted to an "F" representation if this is possible with the given data space. The length of the fractional part is modified accordingly from 0 to (w - 2). If w is less than seven, values smaller than the least significant digit of the fractional string are represented by a string of zeros (which applies also to the "F" format). Too large numbers are indicated by a "format overflow", a string of asterisks. For values of w greater than or equal

to seven, however, the routine changes to scientific notation (see example), using again all w character positions.

Iw.m: The variable indicated by the routine call parameter is interpreted as an INTEGER and treated accordingly. The value w (w > 0) indicates the reserved output string length, and m is a mode selection parameter. A format overflow is indicated by w asterisks. The mode selection parameter represents the type of the INTEGER variable to be converted; it may assume the following values:

m = 0: Signed two-byte INTEGER*2 (-32768 ≤ I ≤ 32767). (In this case, m may be omitted.)

m = 1: Signed one-byte INTEGER*1 (-128 ≤ I ≤ 127).

m = 2: Unsigned two-byte integer (not supported by FORTRAN) (0 ≤ I ≤ 65535).

m = 3: Unsigned one-byte integer (not supported by FORTRAN) (0 ≤ I ≤ 255).

The latter two values of m permit the output of integer data generated by assembly language and PL/M routines rather than FORTRAN.

Xw: A string of w spaces is output to the console. This command can be used in order to clear single lines or parts of lines. Note that, although no variable is involved in this operation, a (dummy) <variable> parameter must be specified which may be a constant or the name of any variable except a CHARACTER variable or constant.

Zw: The w/2 bytes indicated by the <variable> parameter are converted to their hexadecimal representation and output as a string of length w. The parameter w must be an even number greater than zero.

OUTDTX has an error detection routine which reports an "OUTPUT ERROR" if an erroneous control string was encountered. Such an error can be caused by a missing parameter within the string or by a parameter which is out of range. Note that the routine does not distinguish between console, printer, and buffer output errors. The erroneous output request is cancelled. The "OUTPUT" (and "INPUT") "ERROR" messages are displayed in the scrolled block if a divided screen is used; otherwise, they are simply added to the last output line.

They are embedded between strings of asterisks and accompanied by a "beep" signal in order to attract the operator's attention. An "OUTPUT ERROR" should never occur in a debugged system; an "INPUT ERROR" may also be caused by (although not very probable) erroneous operator actions during data entry.

Sample program sequence:

```
        CHARACTER*11 STRING
        CHARACTER*16 COMMD
        INTEGER*1 K
        INTEGER*1 OUTBUF(66)
        INTEGER*2 L
        OUTBUF(1) = 64
        STRING = 'Output line'
        COMMD = '5,40,I5'
        X = 3.141592654
        Y = 123.456789
        K = 99
        L = 4321
        ...
101     CALL FRDATO ('5,30,F7.2', X)
102     CALL FRSTRO ('0,1', STRING)
103     CALL FRDATO ('5,10,E16.7', Y)
104     CALL FRDATO ('99,0,I3.1', K)
105     CALL FRSTRO ('5,48','This is a sample')
106     CALL FRDATO (COMMD, L)
201     CALL FRDTPR ('0,30,F7.2', X)
202     CALL FRDTPR ('0,10,E16.7', Y)
203     CALL FRSTPR ('0,48','This is a sample')
204     CALL FRDTPR (COMMD, L)
205     CALL FRSTPR ('0,1', STRING)
206     CALL FRDTPR ('1,0,I3.1', K
301     DO 302 I=3,66
302     OUTBUF(I)=#20H
303     CALL FRSTBO ('0,1','And one more sample',OUTBUF)
304     CALL FRDTBO ('0,30,G7',X,OUTBUF)
305     CALL FRDTBO ('0,0,F7.0',Y,OUTBUF)
306     CALL FRSTBO ('1,0',' That''s the end.',OUTBUF)
```

The above sequence writes data into the fifth line on the CRT screen and to the bottom of the scrolled screen area, it produces two output lines on the printer, and one line of buffer output. Line 5 on the console screen - as set by statements 101, 103, 105, and 106 - will read (the left margin of the output line is at the left margin of the paper):

```
1.2345679E+02       3.14   4321   This is a sample
```

The last line of the scrolled portion of the screen will be output when statement 104 is processed; it will be:

Output line 99

Note that the output in line 5 will appear on the screen strictly in the order of the pertinent statements. It will, however, remain on the screen until either part of it is overwritten by other output or until the screen is cleared.

The output on the printer will consist of two lines which look exactly like the above two lines. Keep in mind that the sequence of the printer output commands cannot be chosen as freely as in the case of the CRT output. Once a line number other than zero was encountered, the output line is printed, and nothing can put any additional output into this line. The first line in our example will be printed when statement 204 is executed, the second, after statement 206.

The user supplied buffer OUTBUF contains 64 actual buffer locations; including the two control locations at its beginning, its total size amounts to 66 bytes. The first byte is set to the length of the buffer proper (64) in the initialization sequence. Lines 301 and 302 overwrite the buffer proper with spaces (#20H). The following lines build an output line within the buffer which will read:

And one more sample 3.14159 123. That's the end.

A carriage-return - line feed pair is appended at the end of the above output. The length of the actually used part of the buffer can be obtained from the second buffer location, in our case, OUTBUF(2). This location holds the number of the next byte after the output string. The total length of a string without a CR-LF is therefore OUTBUF(2) - 1, and OUTBUF(2) + 1 if a CR-LF pair was added.

(The statement numbers in the sample program were introduced for reference purposes only.)

The following examples show the results of the different conversion routines for REAL variables (the arrows indicate the widths of the reserved areas):

VARIABLE	E12.4 <---------->	F12.4 <---------->	G12 <---------->	G5 <--->
0.	0.	0.	0.	0.
1.E-20	1.0000E-20	.0000	1.00000E-20	.0000
-1.E-20	-1.0000E-20	-.0000	-1.00000E-20	-.000
1.23456E-6	1.2346E-06	.0000	.00000123456	.0000
3.141592654...	3.1416E+00	3.1416	3.1415926500	3.142
-3.141592654...	-3.1416E+00	-3.1416	-3.141592650	-3.14
9999.49	9.9995E+03	9999.4900	9999.4900000	9999.
9999.50	9.9995E+03	9999.5000	9999.5000000	****.
1.E6	1.0000E+06	1000000.0000	1000000.0000	****.
-1.E6	-1.0000E+06	*******.****	-1000000.000	****.
1.E10	1.0000E+10	*******.****	10000000000.	****.
-1.E10	-1.0000E+10	*******.****	-1.00000E+10	****.
1.E20	1.0000E+20	*******.****	1.00000E+20	****.
pos. overflow	+*.****	+*******.****	+**********.	+***.
neg. overflow	-*.****	-*******.****	-**********.	-***.
indefinite	?*.****	?*******.****	?**********.	?***.
	<---------->	<---------->	<---------->	<--->

Note: The areas reserved for the output of an item are overwritten in any case when the output action is performed, even if they partly consist of spaces. Although other data may be written into these blank regions, this output will be destroyed the next time the previous item is output.

ROUTINE FRDATO:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Required Stack: 14 bytes

ROUTINE FRSTRO:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Required Stack: 14 bytes

ROUTINE FRDTPR:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Required Stack: 14 bytes

ROUTINE FRSTPR:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Required Stack: 14 bytes

ROUTINE FRDTBO:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Required Stack: 14 bytes

ROUTINE FRSTBO:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Required Stack: 14 bytes

5.2.2.4 I/O Mode Selection and Auxiliary Routines

The following routines permit to select certain features of the I/O tasks INDATX and OUTDTX. They may be called at any time from an application task.

5.2.2.4.1 Input Mode Selection Routine FRINMD

This routine permits to specify whether or not an input line is to be echoed to the scrolled portion of the console screen in the split screen mode. This command is, however, ineffec-

tive if a completely scrolled screen is being used. Echo output may also be suppressed for certain input actions if the <flag> value in the FRDATI or FRSTRI call is set accordingly.

ROUTINE FRINMD:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Routine Call:

```
CALL FRINMD (flag)
```

with: flag: Integer parameter:
flag = 0: no echo output
flag <> 0: echo output generated

Required Stack: 14 bytes

5.2.2.4.2 Output Mode Selection Routine FROUTM

The routine FROUTM permits to switch between the completely scrolled and the split screen modes. For the latter, the number of the first line of the scrolled part and the number of lines within the scrolled part must be specified.

ROUTINE FROUTM:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Routine Call:

```
CALL FROUTM (line,length)
```

with: line: Number of the first line of the scrolled block (for split screen mode) or zero (for completely scrolled screen)
length: Number of lines within the scrolled block (irrelevant for line = 0)

Required Stack: 14 bytes

5.2.2.4.3 Printer Mode Selection Routine FRPRMD

A call to this routine switches the printer off and on under software control. In addition, switching the printer on with this routine enables the printer output again if it was disabled because the printer was found inoperable by OUTDTX.

ROUTINE FRPRMD:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Routine Call:

```
CALL FRPRMD (flag)
```

with: flag: Integer parameter:
flag = 0: no printer output
flag <> 0: printer output generated

Required Stack: 14 bytes

5.2.2.4.4 Input Prompt String Selection Routine FRINPR

This routine allows to specify a prompt string which may be output at the beginning of the input line. This feature permits, for example, to inform the operator about the current status of the system, e.g., about the current command level, and it allows even to explicitly request data. An arbitrary string consisting of printable and non-printable characters may be chosen. The string length permitted depends on the cursor addressing mode of the console terminal used; for a terminal with four byte cursor addressing codes and two byte relative positioning and line clearing codes, the string length is limited to 22 characters; longer strings are truncated. In order to delete the prompt string, the value (not the character!) 0 must be specified as a parameter. (In order to avoid undue software overhead, FRINPR is not used in the CGCS.)

Note: A FROUTM call clears the input prompt string. In order to set a certain output mode and to specify a (printable) input prompt, first the FROUTM call, and afterwards the call to FRINPR has to be issued!

ROUTINE FRINPR:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Routine Call:

```
CALL FRINPR (character)
```

with: character: CHARACTER*1 variable or single character string holding the new input prompt character; value 0 (or ASCII NUL character) for clearing the input prompt string.

Required Stack: 14 bytes

5.2.2.4.5 Screen Clearing Routine FRCLRO

There are two ways to delete output on a split screen: either can the screen be cleared line by line, using an "X80" format (for an 80 character wide screen) in an FRDATO command, or a single FRCLRO call is performed. The first approach is useful if only part of the screen is to be erased; in order to blank the screen completely, however, the FRCLRO call is by far more efficient. Note: Although FRCLRO erases the output on the screen, it does not clear the scroll buffer. The contents of the scrolled block will therefore appear again on the screen when the next output to the scrolled area is performed. The only way to clear the scroll buffer is to write blank lines (one for each scroll block line) to it.

ROUTINE FRCLRO:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call.

Routine Call:

```
CALL FRCLRO
```

Required Stack: 14 bytes

5.2.2.4.6 Printer Timeout Setting Routine FRSPTO

This routine permits to set the printer timeout (in iRMX-80 time units of 50 ms) to a value differing from the one specified at program linkage time. This permits, for example, to increase the printer timeout if large amounts of data are to be printed and if the printer was already found operable. FRSPTO can be called at any time by any task within the system.

ROUTINE FRSPTO:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none.

Routine Call:

```
CALL FRSPTO (timeout)
```

with: timeout: INTEGER*2 variable or constant specifying the desired printer timeout in iRMX-80 time units (50 ms).

Required Stack: 0 bytes

5.2.2.4.7 Output Mode Change Indicator Function FRMCHG

This function must be declared LOGICAL*1 within the calling FORTRAN routines. FRMCHG returns a .TRUE. value only when a background system terminated its operations, and when the foreground system using OUTDTX routines will probably have to clear and restore its screen (compare chapter 3.3.4.2). (Background systems can be invoked with "Cntl-C" via the Alternative Terminal Handler.) Since there is no background system in the CGCS, FRMCHG is not needed there.

ROUTINE FRMCHG:

Routine Type: Assembly language subroutine; reentrant; must be declared as LOGICAL*1 in the calling FORTRAN program.

Initialization: none.

Routine Call:

boolean = FRMCHG (dummy)

with: boolean: LOGICAL*1 variable (or immediate use of FRMCHG as parameter, e.g., in logical IF statements).
dummy: arbitrary variable or constant (no CHARACTER!).

Required Stack: 0 bytes

5.2.2.5 Control String Building Routine FRCSTR

The control strings for OUTDTX are primarily defined when the source program containing the output function calls is written. In order to permit the definition of these control strings at runtime, under program control, the CHARACTER*16 function FRCSTR was provided. This function converts two integer parameters and a string (or a previously defined CHARACTER variable) to a string which is accepted by the OUTDTX routines. Output lines and/or columns may therefore be selected directly by the program software. FRCSTR can advantageously be used as a parameter in an output routine call.

ROUTINE FRCSTR:

Routine Type: Assembly language subroutine; reentrant; must be declared as a CHARACTER*16 variable in a FORTRAN program.

Initialization: none

Routine Call:

character variable = FRCSTR (line,column,format)

with: line: Integer constant or variable, holding the line number (compare chapter 5.2.2.3).
column: Integer constant or variable, holding the column number (compare chapter 5.2.2.3).
format: Format string (or CHARACTER variable holding a format string) (compare chapter 5.2.2.3)

Required Stack: 6 bytes

5.2.2.6 Auxiliary Routines

The following routines do normally not require the programmer's attention. They are subroutines which are called by the I/O tasks INDATX or OUTDTX. They may, however, be used by other routines than those contained within the I/O libraries. Still, their use requires great care as some of them are neither reentrant nor protected.

Six conversion routines are kept in the library FORTIO.LIB. The library NOFLOT.LIB provides the hexadecimal and decimal integer I/O routines only, and ties away to dummy subroutines the references to the floating point I/O routines. (This library can therefore be used for all applications which do not require floating-point I/O.) All conversion routines were written in assembly language; they can be called by assembly language routines only since the high speed requirements imposed demanded a more efficient parameter passing than possible with FORTRAN or PL/M. In the following, only a summary of the parameters required for calling them from an assembly language routine is given. The non-reentrant routines FXFLIN and FXFLOT must not be shared between the routines in FIORMX.LIB and any application software. If they are required elsewhere within a system which also contains the I/O tasks discussed in this chapter, a separate copy of them must be supplied. These restrictions do not apply, however, to the other four routines which are reentrant.

ROUTINE FRSTHX:

Routine for the conversion of ASCII strings into positive INTEGER*1 variables (modulo (128)).

PARAMETERS:

A ... Result (O)
C ... Input string counter (remaining string length + 1) (I,O)
D+E . Input string pointer (I,O)

Note: D+E point to the character after the next non-blank or non-digit on return!

ROUTINE FRFXIN:

Routine for the conversion of ASCII strings into INTEGER*2 variables.

PARAMETERS:

C ... Input string counter (remaining string length + 1) (I,O)
D+E . Input string pointer (I,O)
H+L . Result (O)

On return, D+E point to the first non-digit which follows a digit. The routine requires an error handler module FXIERR. The corresponding features of INDATX apply.

ROUTINE FXFLIN:

Routine for the conversion of ASCII strings into REAL variables.

PARAMETERS:

STACK Address for the storage of the result (I)
C ... Input string counter (remaining string length + 1) (I,O)
D+E . Input string pointer (I,O)

On return, D+E point to the first non-digit which follows a digit, a sign, or an "E". The routine requires an error handler routine FXIERR. The corresponding features of INDATX apply.

ROUTINE FRHXOT:

Routine for the conversion of a (binary) byte into two bytes of ASCII-coded hexadecimal representation.

PARAMETERS:

C ... Byte to be converted (I)
D+E . Pointer within the output buffer (I,O)

On return, D+E point to the location within the output buffer which follows the last character converted. The register pair H+L is not used by the routine.

ROUTINE FRFXOT:

Routine for the conversion of INTEGER*1 or INTEGER*2 variables into ASCII strings.

PARAMETERS:

STACK Address of the integer which is to be converted (I)
STACK Start address of the output string (I)
C ... Mode: (I)
0 ... Signed two-byte integer (INTEGER*2)
1 ... Signed one-byte integer (INTEGER*1)
2 ... Unsigned two-byte integer
3 ... Unsigned one-byte integer
E ... Length of the output string (I)

The features discussed for OUTDTX apply analogously.

ROUTINE FXFLOT:

Routine for the conversion of REAL variables into ASCII strings.

PARAMETERS:

B+C . Start address of a 7 byte control area in memory:
Byte 0-1: Address of the REAL variable to be converted
Byte 2-3: Start address of the output string
Byte 4: Format type (ASCII character):
"E" ... Scientific notation
"F" ... Floating-point format
"G" ... Hybrid format
Byte 5: Output string length
Byte 6: Fractional part length

The features discussed for OUTDTX apply analogously.

5.2.2.7 ISIS-II and RXISIS-II Versions of the I/O Routines

Three libraries, FIOISS.LIB, FIORXI.LIB, and FIORXR.LIB, are provided in addition to the "standard" FIORMX.LIB library to permit the execution of the above I/O routines under ISIS-II and under RXISIS-II. They are, accordingly, used by the ISIS-II or RXISIS-II-based auxiliary routines which support the CGCS, e.g., by the Macro Command Editor COMMED. In general, the properties of the iRMX-80 routines are reduplicated within

the ISIS-II and RXISIS-II versions, with the exception of some genuine real-time functions. There are two different versions for an RXISIS-II environment: The routines in FIORXR.LIB behave like the ISIS-II routines in FIOISS.LIB, while the routines in FIORXI.LIB essentially reduplicate the features of the iRMX-80 routines in FIORMX.LIB. The following table shows the major differences between the four libraries:

LIBRARY	FIORMX	FIORXI	FIORXR	FIOISS
environment	iRMX-80	RXISIS-II		ISIS-II
ROM vsn dependence	YES		NO	
initialization	FRIOST	FRINIO		
input prompt	0 - 22 CHARACTERS		0 - 1 CHARACTER	
printer timeout	YES	NO		
FRSPTO	YES	NO		
FRMCHG	YES		NO	
exit at CNTL-Z	NO	YES		

The following essential differences apply:

* The ISIS-II and RXISIS-II routines must be initialized by a call to the subroutine FRINIO which does not take any parameters. The FRIOST call of the iRMX-80 routines is, in contrast, not required.

* There is no timeout for the printer under ISIS-II or RXISIS-II. Programs will "freeze" if printer output is requested while the printer is not ready. No error message is generated in this case either.

* There is no restriction as to the line editing features of ISIS-II or RXISIS-II. In particular, all control characters of the Alternative Terminal Handler are available under both RXISIS-II library versions; there is, however, no Output Mode Change Detection (routine FRMCHG) in FIORXR.LIB. With the FIOISS.LIB and FIORXR.LIB routines, the use of the Cntl-R and Cntl-X commands should be avoided if the input echo line exceeds one line on the CRT. Using these commands in this case will mess up the display. Note that "Escape" does not delete the input

line display on the CRT screen although it deletes the contents of the input line. With all libraries except FIORMX.LIB, the entry of Cntl-Z terminates the execution of a program. An appropriate sign-off message is provided by the I/O routines.

In contrast to the iRMX-80 or RXISIS-II based I/O routines which use the cursor positioning routine of the Alternative Terminal Handler, a cursor positioning routine must be specially provided for the ISIS-II based routines. The standard cursor positioning routine uses a step-by-step motion of the cursor in order to position it to the current output position. Although this is the only approach which is compatible with most terminals (including the old older versions of Intellec development systems), it is not optimal as it requires long output times as well as a large buffer. If the terminal used has the capability of direct cursor motion, an alternative FRPSCR routine should be used. This routine should be linked in prior to FIOISS.LIB when the software is configured. An alternative FRPSCR routine can be designed according to the rules given in chapter 3.3.4.1.7 for the cursor positioning routine of the Alternative Terminal Handler. Defining an alternative cursor positioning routine will usually not affect the stack requirements of the I/O tasks. (More than 100 bytes of the stack of OUTDTI are not used at the time of the FRPSCR call, and any such routine is not likely to require more than a small fraction of this available stack area).

In order to fully utilize the advantages of an alternative FRPSCR routine, the parameters which are normally kept in the module FXTISS (which would have to be changed anyhow if a console terminal other than an Intel system is used) should also be declared PUBLIC by this routine (compare chapter 5.2.2.8). While all parameters in FXTISS can be changed in a rather straightforward way, some considerations should be applied to the length of the transfer buffer F0TRBF as the size of this buffer can be considerably reduced if a terminal with direct cursor addressing is used. Its length can be calculated as follows:

```
MAX. LENGTH OF THE CURSOR POSITIONING STRING +
CRT SCREEN WIDTH (CHARACTERS PER LINE) +
CURSOR POSITIONING STRING (FOR LAST LINE, COLUMN 1) +
(24 * NUMBER OF LINES IN THE INPUT AREA) - 1 +
LENGTH OF THE INPUT BUFFER OF THE TERM. HANDLER (122)
```

5.2.2.8 Configuration Constants Used by the I/O Routines

The strong dependence of the I/O routines on the hardware on which they are executed prevented their completely straightforward insertion into the application code. Several data modules are required for all environments. Standard data which apply to an Intellec development system are kept in library modules in FIOISS.LIB and will be inserted into the final code unless different data are explicitly linked in in front of the library files. A similar approach is used for the iRMX-80 and RXISIS-II based routines. While the general features of program linkage will be discussed in chapter 6 of this documentation, the special data files dedicated to the I/O programs are presented below. The following table lists all PUBLIC variables required by the I/O routines and, if applicable, their default values. A "D" in the column "Default Value" indicates that memory locations in the data segment are assigned to the particular PUBLIC label (via a "DS" assembler directive) rather than a value (with a "SET" or "EQU" directive). The first two PUBLIC variables shown in the table are, for example, defined by means of the assembly language code sequence:

```
        PUBLIC  F0IBFL,F0IBUF
F0IBFL  SET     80
        DSEG
F0IBUF: DS      F0IBFL
        END
```

Note that all PUBLIC variables within a module must be specified in an alternative module. The following source files which contain one parameter module each may be modified in order to provide alternative modules:

MODULE	FIORMX.LIB	FIORXI.LIB	FIORXR.LIB	FIOISS.LIB
FXCONF	FXCRMX.SRC	FXCRXI.SRC	FXCRXR.SRC	FXCISS.SRC
FXOFST	FXOFST.SRC	-	-	-
FXTERM	FXTRMX.SRC	FXTRXI.SRC	FXTRXR.SRC	FXTISS.SRC
FXPRDT	FXPRMX.SRC	FXPRXI.SRC	FXPRXR.SRC	FXPISS.SRC

MODULE	VARIABLE	DEFAULT	MEANING
FXCONF	F0IBFL	80/122	Input buffer length @)
	F0IBUF	D	Input buffer with length F0IBFL
FXOFST	F0OFST	18	Offset value depending on the FORTRAN floating-point routines (see 5.2.2.1) *)

FXTERM	F0LINC	24/25	Number of console output lines @)
	F0COLC	80	Number of output columns on the CRT
	F0MXSC	8	Maximum number of scrolled lines in split-screen mode
	F0INLC	3	Number of lines in input area
	F0CURU	1B41H	Cursor up (Esc-A) #)
	F0CURD	1B42H	Cursor down (Esc-B) #)
	F0CURL	1B44H	Cursor left (Esc-D) #)
	F0CURR	1B43H	Cursor right (Esc-C) #)
	F0CURH	1B48H	Cursor home (Esc-H) #)
	F0CLRS	1B45H	Clear screen (Esc-E) #)
	F0CLRL	1B4BH	Clear line (Esc-K) #)
	F0LMIC	F0LINC-F0INLC	
	F0DMYS	(F0INLC-1)*24	
	F0TRBF	D	Transfer buffer +)
	F0OBUF	D	Line output buffer, length F0COLC
	F0SBUF	D	Scroll buffer, length F0COLC*F0MXSC
	F0CRBF	D	Auxiliary array, length F0MXSC*3
FXPRDT	F0PRBL	120	Printer buffer length
	F0PRBF	D	Printer buffer, length 2*F0PRBL
	F0PRTO *)	40	Printer timeout (in RMX time units)
	F0PRTM *)	10	Number of "Printer not ready" mssgs

@) RMX-80 AND RXISIS-II / ISIS-II
*) FIORMX.LIB only
#) FIOISS.LIB only

+) Length of F0TRBF:
iRMX-80 AND RXISIS-II:
8+F0COLC

ISIS-II: 2*(2+(F0LINC-1)+F0COLC) + 2*(1+(F0LINC-2))+ F0INLC*24-1 + F0IBFL
(cursor pos. + output) + (input area prep.) + (inp. echo)

5.2.2.9 CGCS-Specific I/O Routines

The specific operation of a process control system like the CGCS requires that some output items, in particular, the entire dialogue between the operator and the system, should also be recorded for documentation purposes either on the printer, or on a disk file. A special set of interface routines was therefore specially prepared for the CGCS which can be called by any task which requires input or generates output, namely, STRIN and DATIN for the input of data to vari-

ables of type CHARACTER and of any other type, respectively, and STROUT and DATOUT for the corresponding output operations.

The input routines DATIN and STRIN echo the entire input line to the documentation output, while the output routines DATOUT and STROUT write simultaneously to the screen and to the documentation output. In either case, each documentation output line is preceded by the actual and the internal system time of its generation. The documentation routines format their output into pages of 56 lines each; each page is headed by a line which holds the run's date, a run identification, and a page number.

The CGCS-specific I/O routines are based on the corresponding routines in FIORMX.LIB, and have to be called essentially with the same parameters hence. DATIN and STRIN may be called only when a new line is actually requested from the Terminal Handler; the <flag byte> for these calls must, however, have bit 1 rather than bit 0 set (compare chapter 5.2.2.2.1). (DATIN and STRIN read the entire input line into a buffer which is copied to the documentation output, and subsequently vector control to FRDATI and FRSTRI, respectively. Since the input line has already been read into the interface routines' input buffer, FRDATI or FRSTRI have to re-scan the line rather than requesting a new one from the Terminal Handler.)

The CGCS-specific routines are not reentrant; critical parts of them must therefore be protected by a software interlock which must be initialized with a call to the routine INIPRT before any task requests access. INIPRT is called (without parameters) from the initialization routine FXUSIN (compare chapter 5.3.1.3).

While there is only one task (namely, the Command Interpreter) which requests input from the console, there are several tasks in the CGCS which create output which ought to be routed to the printer or to the Documentation file. This prohibits the use of some of the output buffering features built into FRDATO and FRSTRO; it is not possible to collect items within the output routines which are to be written to one line in the scrolled portion of the console screen, since another task which generates output while a line is being built would obviously interfere with the data already in the buffer. All lines to be written to the scrolled area which could not be generated in a single output command have therefore to be constructed in a buffer with the buffer output routines; only when the line image in the buffer has been completed, it may be output with DATOUT. (Since each task can own a private output buffer, there is no danger of interference any more.)

The peculiarities of a real-time process control system require an extremely high degree of fault tolerance, particularly for the I/O routines. The failure of a peripheral (and, possibly, only auxiliary) device like a printer must by no means permanently detain the operation of the remainder of the system. Therefore, a printer timeout feature was provided which discards printer output if the printer did not respond within a given period (currently, 10 seconds); after three unsuccessful attempts to write to the printer, printer output is disabled altogether. (A corresponding error message is displayed on the CRT console.) Printer output can be activated (or, re-activated) with a call to the subroutine STARTP.

5.2.3 Disk Interface Routines - Libraries FXDISK.LIB and FXDSKI.LIB

The following routines permit the use of basic disk functions by a FORTRAN program without involving the tremendous code overhead imposed by the standard FORTRAN routines. While FXDISK.LIB contains a version for a genuine iRMX-80 environment, the routines in FXDSKI.LIB can be executed under ISIS-II or RXISIS-II. Both versions behave identically with regard to their programming interfaces.

NAME	TYPE	FUNCTION	CHAPTER
FROPEN	subr	disk file opening routine	5.2.3.1
FRREAD	subr	read data from disk file	5.2.3.2
FRWRTE	subr	write data to disk file	5.2.3.3
FRSEEK	subr	perform SEEK operation	5.2.3.4
FRCLSE	subr	disk file closing routine	5.2.3.5
FRLOAD	subr	load code from disk file	5.2.3.6
FRATTR FRDELT FRRNME	subr subr subr	disk file attribute setting disk file deleting routine disk file renaming routine	5.2.3.7
FREXIT	subr	exit to operating system	5.2.3.8
FRDSTA	func	check the status of a disk I/O operation	5.2.3.9
FXDSKE	subr	disk error message generation	5.2.3.10

In contrast to the console and printer I/O routines, the disk I/O operations are confined to an unconverted transfer of strings or binary data. This approach results in a higher transfer speed and in reduced disk space requirements. If necessary, conversions to ASCII can be carried out with the Buffer Output or Input routines described in chapter 5.2.2 prior to a disk file output or after the input from a disk file.

File access for the READ/WRITE/SEEK/CLOSE operations is controlled by a file number which can be freely assigned (as an

INTEGER*1) with the FROPEN call. All further operations refer to the file only by means of the file number. The file name specified with the FROPEN call may define a genuine disk file (in the standard ISIS-II notation), or one of the I/O devices supported by the current operating system. Any valid ISIS-II device may thus be used in an ISIS-II environment; under iRMX-80 (with the Alternative Terminal Handler installed) and RXISIS-II, the a restricted range of devices is supported (compare chapters 3.4.1.1 and 5.2.3.1). In the iRMX-80-based version (FXDISK.LIB), there is no restriction (except the memory available) to the number of concurrently open files if the buffers required by the Disk File System are built in memory supplied by the Free Space Manager. (It is possible to specify in the iRMX-80 Configuration Module whether Free Space Manager supplied or fixed memory locations are to be used for the disk file buffers.) In contrast, the number of concurrently open files is limited to six with the ISIS-II/RXISIS-II version (FXDSKI.LIB). Any valid INTEGER*1 value (-128 to 127) may be used as a file number; all concurrently open files must, of course, have different file numbers. (The use of negative file numbers is, however, not recommended. The Console Input routines described in chapter 5.2.4.2 use file number -1 which should therefore not be used otherwise.)

While the ISIS-II/RXISIS-II version of the Disk Interface Routines (FXDSKI.LIB) is a relatively simple subroutine interface to the corresponding ISIS-II system routines (or to the ISIS-II emulation within RXISIS-II), the iRMX-80-based routines in FXDISK.LIB are considerably more elaborate in order to maintain the full real-time facilities of iRMX-80: Similar to the I/O software covered by the preceding chapter, the disk interface software consists of one central task (FXDISK) which receives messages from small subroutines which are called by the user FORTRAN program. It advances, in turn, messages to the iRMX-80 Disk File System. The message transfer between FXDISK and the subroutines which are called by the user task is essentially identical to the approach used for the I/O system, and the (physically) same message locations are used. This does not impose any disadvantage as a user task may only perform console or disk I/O at a given time. Any user task which executes any kind of disk access via FXDISK must therefore be initialized by an FRIOST call prior to any disk routine call. The same considerations about the task configuration apply as specified in chapter 5.2.2.1, particularly with regard to the "extra" bytes following the task descriptor.

FXDISK requests the appropriate operations from the Disk File System, using the (modified) user task supplied request message, and can subsequently handle the next disk I/O request issued by another task. An auxiliary task receives the Disk

File System responses and releases the request messages to their source tasks. I/O requests are thus "pipelined" through FXDISK, the appropriate Disk File System task, and the auxiliary response task, which implies that several requests may be handled in parallel. The full real-time capabilities of the Disk File System are therefore maintained by the Disk Interface routines. The user task which has requested a disk operation is kept waiting until the operation has been terminated. This is essential as the task may neither be allowed to change data locations before or while their contents are written to disk, nor to continue its processing without knowing the results of the disk operation. This fact excludes, of course, high-speed tasks or tasks with a critical timing from disk operations.

Under iRMX-80, FXDISK maintains a 16 byte control block for each open file which contains the number of the file, its name, the address of the entry exchange supplied by the Disk File System for the particular file, and two auxiliary bytes which contain the link information to the next control block. These internal control structures are built of Free Space Manager memory.

In either version - FXDISK.LIB and FXDSKI.LIB -, FXDISK returns a two-byte "status" value which must be interpreted by the user task. No error check - except those which are required for the internal operations of FXDISK - is performed, and no error message output is generated on the console. This was done on purpose as some applications might involve deliberate disk errors which should not confuse the output on the console. The responsibility for the interpretation of the "status" word remains fully with the application program. Generally, no further disk I/O action should be performed by the calling task until the result of the preceding disk access was checked. A reentrant interface routine - FRDSTA - which must be declared as a LOGICAL*1 function in FORTRAN makes the interpretation of the "status" word easier, and the routine FXDSKE provides an error message on the console unless the application takes error handling actions of its own. (FXDSKE generates only an error message; it does not off-load the calling task from providing some kind of error routine to which it can branch in the case of a disk error.)

TASK NAME:	FXDISK
ENTRY POINT:	FXDISK
STACK LENGTH:	38 bytes
PRIORITY:	higher than all tasks requesting disk operations
DEFAULT EXCH.:	none
EXTRA:	0
INITIAL EXCH.:	FXDSKX

5.2.3.1 Disk File Opening - Routine FROPEN

Prior to any access to a disk file, this file has to be opened by means of a call to the subroutine FROPEN. This call must contain an arbitrary but exclusive INTEGER*1 file number which will also be used in order to identify the file in all future accesses, its file name (in ISIS-II format), and an access parameter which defines the type of file access. The access parameter is an INTEGER number which may assume the values 1, 2, or 3, corresponding to opening for reading, writing, and updating (reading and writing), respectively. Under iRMX-80, a file may be opened for reading under more than one (different) file numbers; still, it may be opened only once for writing or updating. (Under ISIS-II, a disk file may be opened only once for any access type.) The file name specified with the FROPEN call may correspond to a genuine disk file, or to any device supported by the resident operating system. All ISIS-II devices are supported under ISIS-II; under RXISIS-II and iRMX-80 (with the Alternative Terminal Handler), the following devices are supported:

```
:CI: ... Console Input
:VI: ... Console Input
:CO: ... Console Output
:VO: ... Console Output
:LP: ... Line Printer
:TO: ... Line Printer
:BB: ... Byte Bucket
```

The Byte Bucket is a dummy device which, as an output device, simply ignores output data. If the Byte Bucket is used as an input device, it returns an empty string (with length zero). :CI: and :VI: may be opened for input only, all other devices (except :BB:), for output only.

FROPEN returns an INTEGER*2 value which indicates the file status; a zero value corresponds to a successfully fulfilled file opening request, while other values indicate some kind of

a disk error (compare chapters 5.2.3.9 and 5.2.3.10, and Appendix 4).

ROUTINE FROPEN:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call (RMX-80 routines in FXDISK.LIB only).

Routine Call:

```
CALL FROPEN (filenumber, filename, access, status)
```

with: filenumber: Arbitrary INTEGER*1 number, dedicated to the corresponding file.
filename: Filename, corresponding to ISIS rules.
access: Integer parameter:
1 ... opened for reading
2 ... opened for writing
3 ... opened for updating
status: Error status parameter (zero for faultless operation, non-zero in the case of an error); compare 5.2.3.10.

Required Stack: 14 bytes.

5.2.3.2 Reading From a Disk File - Routine FRREAD

Each call of this routine transfers a number of bytes to locations in memory whose start address and count have to be given as parameters. The routine returns the number of the bytes actually read which is usually identical to the requested length unless the end of the disk file or a disk error were encountered. Since the end of file is not reported with a non-zero status value, the "actual" value should be checked by the application software, in addition to the "status" parameter. Similar to FROPEN, a two-byte status parameter is used to indicate possible errors.

ROUTINE FRREAD:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call (RMX-80 routines in FXDISK.LIB only).

Routine Call:

```
CALL FRREAD (filenumber,variable,length,actual,status)
```

with: filenumber: see 5.2.3.1 (FROPEN)
variable: Start address for the storage of the data read from the disk file.
length: Number of bytes to be read from the file.
actual: Number of bytes actually read from the file.
status: see 5.2.3.1 (FROPEN)

Required Stack: 14 bytes.

5.2.3.3 Writing To a Disk File - Routine FRWRTE

Each call to this routine transfers a number of bytes from memory locations whose start address and count have to be given as parameters to a disk file indicated by its file number.

ROUTINE FRWRTE:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call (RMX-80 routines in FXDISK.LIB only).

Routine Call:

```
CALL FRWRTE (filenumber, variable, length, status)
```

with: filenumber: see 5.2.3.1 (FROPEN)
variable: Start address of the data to be written to the disk file.
length: Number of bytes to be written to the file.
status: see 5.2.3.1 (FROPEN)

Required Stack: 14 bytes.

5.2.3.4 Access to Random Files - Routine FRSEEK

The routines FRREAD and FRWRTE permit the input from and the generation of conventional sequential files. Random disk files may be handled with these routines, provided that the file marker (which indicates the position of the block which is to be read or written within the file) is moved to the correct position prior to the actual read/write operations. The file marker can be positioned by means of the routine FRSEEK. An extensive description of the SEEK function can be found in the ISIS-II or iRMX-80 User's Guides.

ROUTINE FRSEEK:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call (RMX-80 routines in FXDISK.LIB only).

Routine Call:

```
CALL FRSEEK (filenumber,mode,blockno,byteno,status)
```

with:
- filenumber: see 5.2.3.1 (FROPEN)
- mode: INTEGER*1 parameter:
  - 0 ... return current marker position
  - 1 ... Decrement marker position
  - 2 ... Set marker to new position
  - 3 ... Increment marker position
  - 4 ... Jump to end of file
- blockno: file block number (0 ... 4004)
- byteno: Byte number within the block (0 ... 127)
- status: see 5.2.3.1 (FROPEN)

Required Stack: 14 bytes.

5.2.3.5 Disk File Closing - Routine FRCLSE

Any access to a disk file must be terminated by closing this file. Disk files opened under iRMX-80 for writing or updating which have not been closed properly do not show up in the disk directory and can therefore no more be accessed. Similar to all other file accessing routines, FRCLSE references the file by means of its number; a status value is returned by FRCLSE.

ROUTINE FRCLSE:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call (RMX-80 routines in FXDISK.LIB only).

Routine Call:

```
CALL FRCLSE (filenumber, status)
```

with: filenumber: see 5.2.3.1 (FROPEN)
status: see 5.2.3.1 (FROPEN)

Required Stack: 14 bytes.

5.2.3.6 Program Loading - Routine FRLOAD

In contrast to the preceding routines which are designed for handling data disk files or I/O from/to physical devices, the subroutine FRLOAD permits to load program code from disk. Under iRMX-80 and RXISIS-II, only genuine disk files may be specified with the FRLOAD call; ISIS-II permits also devices as a source of code loading operations.

FRLOAD loads code into the system's read-write memory without transferring control to this code; its basic function is therefore loading subroutine overlays which are eventually invoked by the resident program code. It is possible to specify a bias value with the FRLOAD call which shifts the program code to memory locations different from the memory area defined during the overlay linkage. The code can usually not be executed in these shifted locations, still, it can be stored there and can be moved later into its correct position. FRLOAD does not prevent main program code from being loaded, which is illegal in a genuine multi-tasking environment under iRMX-80; still, an error message (a non-zero status value) is returned if the loaded code was a main program. The application program has to make sure not to access the loaded code in the case of a disk error.

ROUTINE FRLOAD:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call (RMX-80 routines in FXDISK.LIB only).

Routine Call:

CALL FRLOAD (filename, bias, status)

with: filename: Filename, according to ISIS-II rules.
bias: Bias value (INTEGER*2), usually zero
status: Error status parameter (zero for faultless operation, non-zero in the case of an error); compare chapter 5.2.3.10 and Appendix 4.

Required Stack: 14 bytes.

5.2.3.7 Directory Maintenance - Routines FRATTR, FRDELT, and FRRNME

The above three routines effect the interface to the directory maintenance functions ATTRIB, DELETE, and RENAME, respectively. They can only be used in conjunction with genuine disk files, not with I/O devices.

ROUTINE FRATTR:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call (RMX-80 routines in FXDISK.LIB only).

Routine Call:

CALL FRATTR (filename, control string, status)

with: filename: Filename, according to ISIS-II rules.
cntl string: CHARACTER*2 string with the form {F|I|S|W}{0|1}, according to ISIS-II conventions.
status: Error status parameter (zero for faultless operation, non-zero in the case of an error); compare chapter 5.2.3.10 and Appendix 4.

Required Stack: 14 bytes.

ROUTINE FRDELT:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call (RMX-80 routines in FXDISK.LIB only).

Routine Call:

```
CALL FRDELT (filename, status)
```

with: filename: Filename, according to ISIS-II rules.
status: Error status parameter (zero for faultless operation, non-zero in the case of an error); compare chapter 5.2.3.10 and Appendix 4.

Required Stack: 14 bytes.

ROUTINE FRRNME:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call (RMX-80 routines in FXDISK.LIB only).

Routine Call:

```
CALL FRRNME (filenameold, filenamenew, status)
```

with: filenameold: Old filename, according to ISIS-II rules.
filenamenew: New filename, according to ISIS-II rules.
status: Error status parameter (zero for faultless operation, non-zero in the case of an error); compare chapter 5.2.3.10 and Appendix 4.

Required Stack: 14 bytes.

5.2.3.8 Exit to Operating System - Routine FREXIT

The routine FREXIT should be used in order to terminate the operation of the current program or real-time system. Upon call to FREXIT, all open files are closed, and control is vectored to the resident system, i.e., to ISIS-II or RXISIS-II if the routines in FXDSKI.LIB are used, or to an appropriate initialization of iRMX-80 for FXDISK.LIB. (In fact, a routine R@EXIT is called in the latter case. The default R@EXIT routine provided in an RXISIS-II based environment will re-boot RXISIS-II.)

ROUTINE FREXIT:

Routine Type: Assembly language subroutine; reentrant.

Initialization: FRIOST call (iRMX-80-based routines in FXDISK.LIB only).

Routine Call:

```
CALL FREXIT
```

Required Stack: 14 bytes.

5.2.3.9 Disk File Status Checking - Function FRDSTA

It has already been mentioned that the above disk file accessing routines do not perform any exception handling of their own if an error condition is detected, except returning a non-zero "status" value. This status value should be checked by the application code after each disk access. Since a frequent check of an INTEGER*2 value with FORTRAN "IF" statements imposes an undue code overhead, the LOGICAL*1 FUNCTION FRDSTA was provided. (The compiler generated code is considerably less extensive if a logical "IF" is applied to a Boolean variable rather than an INTEGER.)

ROUTINE FRDSTA:

Routine Type: Assembly language subroutine; reentrant; must be declared as LOGICAL*1 in the calling FORTRAN program.

Initialization: none

Routine Call:

```
boolean = FRDSTA (status)
```

with: boolean: LOGICAL*1 variable (or immediate use of FRDSTA as a parameter, e.g., in a logical IF statement).
status: Error status parameter (zero for faultless operation, non-zero in the case of an error); compare chapter 5.2.3.10 and Appendix 4.

Required Stack: 0

5.2.3.10 Disk Error Message Generation - Routine FXDSKE

A disk error reported by the disk file handling routines does not necessarily mean that there was actually an error condition. In order to determine, for example, whether a disk file with a certain name already exists, e.g., to permit its protection from an inadvertent destruction by overwriting, the file can first be opened for reading rather than writing. There will be an error reported, of course, if a file with the specified name does not yet exist, although this condition is actually the error-free case. Generally, there are three different possibilities in treating a non-zero status value returned by the disk handling routines:

* The status parameter did, indeed, not indicate an error; it can therefore be ignored.

* The application program outputs an error message of its own, in addition to actions related to the fact that the last disk operation was not successfully performed.

* The generic error message output provided with FXDSKE is used. The application software has to branch according to the faulty disk operation.

The routine FXDSKE requires the software background and support of the output routines described in chapter 5.2.2.3. It generates the following error message in the scrolled part of the console output:

```
***** DISK ERROR xxx yy (TASK tsknam, LOC hexl) *****
```

This line is accompanied by a "beep". The task name and error location information is treated identically to the system

error message described in chapter 5.1.1.6. An error number ("xxx") is provided in order to identify the particular error condition. FXDSKE may be called after each disk access, regardless whether the status value actually reported an error or not. The routine is immediately skipped if the status parameter was zero. Refer to Appendix 4 for a complete list of error messages.

The error message generation routine FXDSKE is not reentrant but protected by a software interlock. This interlock is initialized by the routine FXDSKI which must be called during system initialization.

ROUTINE FXDSKE:

Routine Type: Assembly language subroutine; not reentrant; protected by a software interlock.

Initialization: FXDSKI call.

Routine Call:

```
CALL FXDSKE (status)
```

with: status: Status parameter returned by the disk handling routines (INTEGER*2).

Required Stack: 16 bytes.

5.2.4 General Utility Routines - Library FXUTIL.LIB

This library contains a set of utility functions which are frequently required in a real-time system. The following subprograms are kept in FXUTIL.LIB:

NAME	TYPE	FUNCTION	CHAPTER
FXTIME FRSETT	task subr	timer task reset timer	5.2.4.1
FXOCNS FXRCNS FXCCNS	subr subr subr	open console file read from console file close console file	5.2.4.2
FRCMPS FRCVUC	func subr	string comparison routine string conversion to uppercase	5.2.4.3
FRPOKE FRPEEK FRADDR	subr subr func	transfer of data to memory transfer of data from memory returns address of parameter	5.2.4.4
FRADD FRMULT FRSHFT	subr subr subr	overflow-protected addition rout. overflow-protected multiplication scaling by powers of 2	5.2.4.5
FRPIDC	subr	PID controller routine	5.3.2.1

The generic PID controller routine FRPIDC will be discussed together with the actual crystal growth control routines in chapter 5.3.2.1; it is, however, part of FXUTIL.LIB.

5.2.4.1 Timer Task FXTIME

FXTIME is a multi-purpose iRMX-80 task which can perform most of the lower-speed timing of an application system. (It does so, indeed, in the CGCS.) Using the on-board clock of the CPU board, it generates flag interrupts (compare chapter 5.1.1.4) each second, every ten seconds, each minute, and in arbitrary programmable intervals from 1 to 256 seconds. Furthermore, it provides two (unsigned) INTEGER*2 seconds counters, one started immediately after the system reset, and one, when a dedicated flag was set. An alarm clock function (linked to a flag interrupt) is executed when the second seconds counter is equal to or exceeds a preset value. Finally, the task gener-

ates ASCII strings (in the format HH:MM:SS) which represent the actual time, the internal system time (i.e., the time since the last system reset), and a relative time which can be started arbitrarily by setting a flag. These three strings are output on the console, once every second, thus off-loading the application program from providing this output; console output can be enabled independently for each of the three time display strings. (The third, relative, time is not displayed in the CGCS.) The interaction between FXTIME and the system is performed exclusively via a 65 byte area in read-write memory which can be regarded as a COMMON block by FORTRAN programs. The start address of this area is declared PUBLIC as F0TIME; it has to be tied to the corresponding FORTRAN COMMON block by means of the procedures discussed in chapter 6.

FXTIME provides a total of five output flags which can be used to trigger a flag interrupt within other tasks, and therefore to control the timing of the system. Three flags - the seconds, ten seconds, and minutes flags - are set in regular intervals, starting with the system reset; the variable interval flag is set in regular intervals which can be defined between 1 and 256 seconds by means of an unsigned INTEGER*1 variable. (Note that values greater than 127 correspond to negative integers in FORTRAN; 128 is represented by -127, and 255, by -1. A zero value causes a 256 seconds interval.) The variable which presets the interval may be changed at any time; still, it does not become effective before the next flag interrupt happened. The fifth flag is set when the alarm clock is triggered, i.e., when the seconds counter #2 which is started when a dedicated flag was set becomes equal to or greater than a preset time. These two time values are stored as unsigned two-byte integers. This approach permitted to extend the executable time range from 32767 seconds (approximately 8 hours) to 65535 seconds; values greater than 32767 are represented by negative INTEGER*2 values in FORTRAN. In addition, the execution time of FXTIME could be cut down significantly by omitting the sign treatment. Since the seconds counter #2 keeps running only while it is enabled by a control flag, there are two ways to disable the "alarm clock": either can the preset time be set to a very high value which is unlikely to be ever reached (which is also done automatically each time an alarm was triggered), or the counter #2 is simply disabled by resetting its control flag. Note: All flags used as Boolean data are single-byte variables which may assume the values 0 (flag reset) or 0FFH (flag set), which correspond to INTEGER*1 values of 0 and -1, and to LOGICAL*1 values of .FALSE. and .TRUE., respectively. Flags which are used as an input are interpreted as reset if all bits of the byte indicated by the address are zero, and as set if any bit

differs from zero, i.e., for any non-zero INTEGER value. In contrast to the "alarm clock" function, the setting of the other four output flags by FXTIME cannot be disabled.

The seconds counter #2 can be enabled at any time; it is reset to zero while it is disabled, starting from zero when the counter is activated again. The counter #1, in contrast, can be reset only by a call to the reset subroutine FRSETT; it can neither be stopped nor disabled. The same considerations with regard to its internal format - unsigned two-byte integer - apply as to the counter #2.

Three character strings hold the display of three different timers: The first timer indicates the time since system reset (or, since the last FRSETT call), the second, the actual time, and the third, a differential time. All three strings have the identical format HH:MM:SS. The internal time wraps around to zero after 96 hours, the actual time is output in a 24 hour format, and the differential time is limited to 99 hours by the two digits display area for the hours. In order to permit a correct display of the actual time, the time of the system reset (or the time when FRSETT was called) has to be made known to FXTIME, which is done by means of three INTEGER*1 variables (for hours, minutes, and seconds, respectively). The differential timer, finally, is reset each time a pertinent flag is set.

These three strings are kept in memory locations which can be accessed via the COMMON block F0TIME; they can also be output on the console. Console output is performed via the I/O routines in FIORMX.LIB which have therefore to be included in the system. Three flags permit the independent activation of the output of each of the strings. The output can be arbitrarily located on the console CRT screen; a control string for the string output routine FRSTRO (compare chapter 5.1.2.3) has to be provided in F0TIME for each timer string.

The 65 byte control area F0TIME contains the following data:

BYTE	TYPE	MEANING
0	I*1	One second interrupt flag byte (0)
1	I*1	Ten seconds interrupt flag byte (0)
2	I*1	One minute interrupt flag byte (0)
3	I*1	Variable interval interrupt flag byte (0)
4	I*1	Alarm clock interrupt flag byte (0)
5	I*2	Seconds counter #1 (from system reset) (0)
7	I*2	Seconds counter #2 (started with byte 34) (0)
9	CH*8	Internal time string (from system reset) (0)

17	CH*8	Actual time string (O)
25	CH*8	Differential time string (O)
33	I*1	Interval for variable interval interrupt (I)
34	I*1	Run flag for seconds counter #2 (I) (0 ... Stop, <>0 ... Run)
35	I*2	Setpoint for alarm clock (counter #2) (I)
37	I*1	Flag: Reset differential timer (<>0 ... Reset) (I)
38	I*1	Time of system reset - Hours (I)
39	I*1	Time of system reset - Minutes (I)
40	I*1	Time of system reset - Seconds (I)
41	I*1	Flag: Enable internal time output (I) (0 ... Disable, <>0 ... Enable)
42	I*1	Flag: Enable actual time output (I)
43	I*1	Flag: Enable differential time output (I)
44	CH*7	Output control string - Internal time (I) (Control string for an FRSTRO call - "<line>,<column>")
51	CH*7	Output control string - Actual time (I)
58	CH*7	Output control string - Differential time (I)

TASK NAME: FXTIME
ENTRY POINT: FXTIME
STACK LENGTH: 34 bytes
PRIORITY: 129 (or even higher)
DEFAULT EXCH.: none
EXTRA: 0

INITIAL EXCH.: none

EXECUTION TIME: 1 ms (worst case) once a second

ROUTINE FRSETT:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none.

Routine Call:

```
CALL FRSETT
```

Required Stack: 2 bytes.

5.2.4.2 Console Input Routines FXOCNS, FXRCNS, and FXCCNS

Three routines - FXOCNS, FXRCNS, and FXCCNS - permit console input from an arbitrary disk file which replaces the console CRT terminal. Replacing the console input by disk file data requires that exactly one logical line must be supplied to the system with each READ call. This is, however, not possible with the standard disk I/O routines of chapter 5.2.3 since these routines return a fixed number of bytes without regarding the logical end of an input line. Therefore, a special Read Console routine FXRCNS was prepared which returns always exactly one logical input line (with a length of 1 to 80 characters). FXRCNS obtains its input either from the console terminal (trivially), or from an arbitrary disk file. A Line Feed Character (0AH) is interpreted as the end of the input line. Input lines exceeding 80 characters (including the terminating CR-LF pair) are truncated to 80 characters; their remainder is submitted with the next FXRCNS call. The input line may be further processed by the User Buffer Input routines FRDTBI and FRSTBI described in chapter 5.2.2.2; the buffer format used is fully compatible.

The console file is opened with a call to FXOCNS; the name of a disk file or device suitable for input must be specified with the call. The file is opened, using the Disk Interface routines of chapter 5.2.3, and assigned the file number -1 (0FFH). In the case of an error during file opening and reading, the error is reported with the default disk error message routine (compare chapter 5.2.3.10), and the console input is re-directed to the console terminal.

An input line is read from the currently valid console file via a FXRCNS call. The start address of an 82 byte buffer according to chapter 5.2.2.2 must be specified with the call:

```
        INTEGER*1 BUFFER(82)
        LOGICAL*1 STAT
        REAL X
C
        CALL FXOCNS ('CONSOL')
C           (a disk file on :F0: with the name CONSOL
C           is to be used.)
100     CALL FXRCNS (BUFFER)
C           (read up to 80 bytes of console input)
        CALL FRDTBI ('1,E',X,BUFFER,STAT)
C           (scan the buffer for a floating-point
C           number and store the result in X)
C
C           (process the input)
C
```

```
        GOTO 100
C           (read the next input line)
```

The above example is equivalent to

```
        LOGICAL*1 STAT
        REAL X
C
        CALL FRDATI ('1,E',X,STAT)
```

if ':CI:' is used in the FXOCNS call rather than 'CONSOL'.

The input from the specified console file is continued until

(a) the console file is explicitly closed with a FXCCNS call, or

(b) the end of the console file is encountered (i.e., if a string of length zero is read). In the latter case, an Error 29 (End of Console Input File) is reported.

In either case, the console terminal is re-opened as the system console.

The Console Input routines are not reentrant, and they are unprotected. They must, therefore, be called by one task only, and only one file can be used in conjunction with them at a given time.

The Console Input routines are not used within the CGCS; they constitute an essential part, though, of the Macro Command Editor utility COMMED (compare chapter 7.2).

ROUTINE FXOCNS:

Routine Type: Assembly language subroutine; not reentrant.

Initialization: none.

Routine Call:

```
CALL FXOCNS (filename)
```

with: filename: Filename, according to ISIS-II rules.

Required Stack: 18 bytes.

ROUTINE FXRCNS:

Routine Type: Assembly language subroutine; not reentrant.

Initialization: FXOCNS call.

Routine Call:

```
CALL FXRCNS (buffer)
```

with: buffer: 82 bytes buffer, formatted according to FRDTBI and FRSTBI rules (compare chapter 5.2.2.2.1).

Required Stack: 18 bytes.

ROUTINE FXCCNS:

Routine Type: Assembly language subroutine; not reentrant.

Initialization: none.

Routine Call:

```
CALL FXCCNS
```

Required Stack: 18 bytes.

5.2.4.3 Command Line Interpreter Support Routines

Command line interpretation involves usually the comparison of input strings with command strings. This comparison cannot be performed in a straightforward way by FORTRAN based software, particularly if the lengths of the input and command strings are not necessarily identical. Two routines, FRCMPS and FRCVUC, have been provided in order to make the command line interpretation easier.

FRCMPS compares two strings which are submitted as parameters, either until the strings are found to differ from each other, until the end of one of the strings was encountered, or until a special "wild card" character was recognized in one of the strings. The routine - which has to be declared as a LOGICAL*1 FUNCTION in FORTRAN - returns a Boolean variable which is .TRUE. if both strings are equal, and otherwise .FALSE.. Since leading blanks (spaces, tabs, and other non-printable characters) are stripped off the strings before the comparison

is performed, the position of a command within a command line does not matter. The "wild card" character is particularly useful if complete words are permitted as commands but an abbreviation of these commands to their leading characters should also be possible. The "wild card" character is defined at system configuration time with the one byte variable F0WCCH. The library FXUTIL.LIB uses a vertical bar ("|") as a default wild card character. (Note: F0WCCH is actually a program constant which can be defined, e.g., with an assembly language "SET" instruction. It is included in the program code rather than being stored in memory like a variable.) Suppose the string "CO|" was specified as one of the two parameters of the function FRCMPS. In this case, a .TRUE. value would be returned if the strings "C", "CO", "COM", "COMMAND", or "CONTROL" were specified as the second parameter. On the other hand, FRCMPS is set to .FALSE. if the string "Cx" is encountered where "x" is any arbitrary number of printable or non-printable characters other than "O". Note that CHARACTER variables are filled up with spaces by the input routines if the input line was shorter than the size of the CHARACTER variable. A single "C" entered on the console and stored in a CHARACTER*4 variable will therefore be followed by three spaces in its internal representation. It will therefore be recognized as different from the above mentioned string "CO|". Only a "C" read to a CHARACTER*1 location will be regarded identical to this comparison string.

Since a correct command which is, e.g., given in lowercase characters would be considered different from an uppercase command string, the subroutine FRCVUC was provided whose commission is to change to uppercase the contents of the CHARACTER variable specified as its parameter. (Actually, characters with an ASCII equivalent of 61H or greater are converted to the range of 41H to 5EH, which affects also some special characters. These characters are rarely used in commands, though.)

ROUTINE FRCMPS:

Routine Type: Assembly language subroutine; reentrant; has to be declared as a LOGICAL*1 FUNCTION in FORTRAN.

Initialization: none

Routine Call:

```
boolean = FRCMPS (string1, string2)
```

with: boolean: LOGICAL*1 variable (or direct use in a LOGICAL IF statement).
string1,2: CHARACTER variables or strings of arbitrary lengths.

Required Stack: 0

ROUTINE FRCVUC:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine Call:

CALL FRCVUC (char.var.)

with: char.var.: CHARACTER variable.

Required Stack: 0

5.2.4.4 Data Transfer To and From Absolute Memory Locations

Complex applications in a real-time system require frequently the direct access to absolute locations in memory, e.g., for the access to Variables in the CGCS. Such an access is impossible directly from a FORTRAN program since FORTRAN permits to handle the value stored at a (symbolically referenced) address only but not the address itself. Two routines, FRPOKE and FRPEEK, permit to store date at a certain address which can be submitted as an INTEGER*2 constant or variable, and to read data from this address, respectively. Therefore, data can be transferred to and from regular FORTRAN variables if their positions in memory are known. Both subroutines require the specification of the numbers of bytes which are to be transmitted; they permit thus not only to move single bytes but also multi-byte variables and even arrays. The number of bytes which may be treated with one FRPOKE or FRPEEK call is limited to 127; negative values of the INTEGER*1 length parameter or zero cause the routine to be skipped without further notice and effect.

The address of a FORTRAN variable can be determined at execution time by means of the INTEGER*2 FUNCTION FRADDR.

ROUTINE FRPOKE:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine Call:

```
CALL FRPOKE (variable, address, length)
```

with: variable: Name of a variable (or first element of an array) which is to be stored in memory.
address: INTEGER*2 value (constant or variable) indicating the start address beginning with which <variable> is to be stored in memory.
length: Positive INTEGER*1 value (>0, <128) indicating the number of bytes to be transferred.

Required Stack: 2 bytes.

ROUTINE FRPEEK:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine Call:

```
CALL FRPEEK (variable, address, length)
```

with: variable: Name of a variable where data are to be stored which are copied from memory.
address: INTEGER*2 value (constant or variable) indicating the start address of the source data.
length: Positive INTEGER*1 value (>0, <128) indicating the number of bytes to be transferred.

Required Stack: 2 bytes.

ROUTINE FRADDR:

Routine Type: Assembly language subroutine; reentrant; has to be declared as an INTEGER*2 function.

Initialization: none

Routine Call:

integer*2 = FRADDR (variable)

with: integer*2: Set to the address of <variable>.

Required Stack: 0

5.2.4.5 Overflow Protected Integer Arithmetics

Three assembly language routines, FRADD, FRMULT, and FRSHFT, permit the overflow protected high-speed integer addition, multiplication, and division operations particularly required for the operation of the generic PID controller routine FRPIDC (compare chapter 5.3.2.1). The results of the FRADD and FRSHFT operations are set to the absolutely largest integer number with the correct sign if they would otherwise exceed the permitted range of an INTEGER*2. FRADD performs the addition of two INTEGER*2 variables, FRMULT, their multiplication to a signed four byte result (INTEGER*4) (in which case an overflow is impossible), and FRSHFT allows to multiply an INTEGER*2 variable by a positive or negative power of 2 which is specified as its second parameter (which corresponds to an appropriate left or right shift of the binary data). Rounding of the least significant bit is provided with FRSHFT in the case of a negative scaling factor, i.e., of a division by a (positive) power of two. With the exception of FRSHFT which can also be called from PL/M programs, these three subroutines can only be invoked by assembly language routines. (The high-speed performance required for these routines prohibited the use of the parameter passing conventions of FORTRAN or PL/M.)

ROUTINE FRADD:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine call: No call from FORTRAN or PL/M!

```
        from assembly language programs:
        CALL FRADD

        Parameters:

            D+E ... First item
            H+L ... Second item, result

    Required Stack: 0


ROUTINE FRMULT:

    Routine Type: Assembly language subroutine; reentrant.

    Initialization: none

    Routine call: No call from FORTRAN or PL/M!

        from assembly language programs:
        CALL FRMULT

        Parameters:
            Input:

            D+E ... First factor
            H+L ... Second factor

            Output:
            D ..... Product, byte 3 (MSB)
            E ..... Product, byte 2
            H ..... Product, byte 1
            L ..... Product, byte 0 (LSB)

    Required Stack: 6 bytes


ROUTINE FRSHFT:

    Routine Type: Assembly language subroutine; reentrant.

    Initialization: none

    Routine call: No call from FORTRAN!

        from PL/M:
        result = FRSHFT (shift, input)

        with:   <result> := <input> * 2 ** <shift>
```

```
        input, result:  type ADDRESS
        shift:          type BYTE

    from assembly language programs:
    CALL FRSHFT

    Parameters:

        C ..... Shift parameter:
                >0 ... Left shift - multiplication
                <0 ... Right shift - division
                0 ... No change
        D+E ... Input value
        H+L ... Result

Required Stack: 0
```

5.2.5 High-Speed Hardware-Based Floating-Point Routines - Library FP8231.LIB

A number of additional numeric routines enhance the performance of FORTRAN programs running on Intel 8080/85 based systems by partly replacing standard FORTRAN modules. Using the hardware Arithmetic Processing Unit (APU) (Intel 8231) rather than software floating-point algorithms, these routines are considerably faster than those provided in the standard FORTRAN-80 libraries, and they require significantly less code. The following interface software was specially prepared for the RXISIS-II/iRMX-80 environment:

* Replacement routines for the standard Intel FORTRAN-80 floating-point algorithms of the libraries FPSOFT.LIB and FPSFTX.LIB. The alternative routines require an Intel 8231 Arithmetic Processing Unit to be present in the system. Programs utilizing the alternative routines must be run either under iRMX-80, or under RXISIS-II.

* Replacement for the most important floating-point functions contained in the standard FORTRAN-80 library FPEF.LIB. The same hardware and environmental requirements apply as above.

* An alternative conversion routine from binary floating-point notation to ASCII strings which replaces the equivalent FPSOFT.LIB or FPSFTX.LIB software. (The output of floating-point numbers is usually performed much more frequently than the complementary conversion from ASCII to floating-point; it appeared therefore not necessary to replace the latter routine by software using the APU.) As above, an 8231 APU and an iRMX-80 or RXISIS-II environment are needed.

The high-speed floating-point routines are used within the CGCS (where they particularly speed up output of numeric data to the console screen), and in the RXISIS-II versions of the support utilities SHODAT, COMMED, and READCM (compare chapter 7.).

5.2.5.1 General Information

The alternative floating-point routines effect a transparent replacement of the standard software floating-point algorithms contained in the Intel supplied FORTRAN-80 libraries. They are based on a dedicated Numeric Processor, namely, the Intel 8231 APU (Arithmetic Processing Unit). They adapt the standard floating-point format used by FORTRAN-80 to the special

format required by the APU, and vice versa. Special provisions were made to extend the relatively limited numeric range of the APU (±1.0E-19 ... ±1.0E19) to the standard FORTRAN-80 range (±1.0E-38 ... ±3.4E38). Despite the considerable software overhead imposed by the different data formats, the alternative routines run significantly faster than their software counterparts. The amount of code required is reduced by about 50 percent; the stack is also significantly smaller. The overall system performance is thus improved considerably.

Aside from two exceptions, the routines in FP8231.LIB need no explicit calls from FORTRAN programs; the proper calls are automatically inserted by the FORTRAN compiler. The implementation approach chosen for FP8231.LIB off-loads therefore the programmer from the burden of explicitly calling APU-based software.

The following features are provided in FP8231.LIB:

* Basic arithmetic routines, corresponding to the FORTRAN-80 routines in FPSOFT.LIB or FPSFTX.LIB:

  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Square
  - Square Root

* Floating-point to ASCII conversion, corresponding to the routine FQFB2D in FPSOFT.LIB or FPSFTX.LIB.

* Transcendental functions, replacing routines in FPEF.LIB:

  - Logarithm (natural and common)
  - Exponent
  - Sine, Cosine, and Tangent
  - Inverse Sine, Cosine, and Tangent
  - Arctangent of two parameters (ATAN2)
  - Absolute Value of a Complex Number (CABS)

The interface routines for the 8231 APU were designed for operation in a system based upon standard Intel OEM Single Board Computer hardware. An 8231 Numeric Processor Multimodule expansion board (iSBX 331) is available for the 8085-based iSBC 80-24 Single Board Computer. With regard to this hardware environment, the software was designed to run under Intel's Real Time Multitasking Executive iRMX-80 (and hence, under RXISIS-II.) Therefore, the APU-based routines

have to meet the specific requirements of a real-time (or, a pseudo-real-time) environment with a multitasking approach.

The main target pursued with the APU interface routines was to replace the standard FORTRAN-80 software floating-point library routines with a set of functionally equivalent software. The standard FORTRAN-80 routines are either reentrant (i.e., they use the stack of the task from which they were invoked as a scratchpad for internal data), or they use resources which are local to each task (namely, the floating-point accumulator which is allocated in an extension of the Task Descriptor of each task which performs floating-point operations). Both approaches protect data handled by the routines from interferences if a routine is interrupted while servicing one task, and eventually invoked by another task without being permitted to finish the previous operation. The APU hardware is, in contrast, a shared resource, and provisions must be made to prevent tasks from interrupting an APU operation in progress. There are, in general, two possibilities to achieve this goal:

* Software interlocks, or

* Disabling interrupts.

The first approach is the one primarily suggested by the structure of iRMX-80, and it appeared initially favorable because it permits the CPU to continue processing while the APU is busy. With regard to the very fast APU operation, however, the overhead to execute the interlock structures would require more time in most cases than the APU action to be protected. The longest APU algorithm needs less than 3 milliseconds (the Power function), and the plain arithmetic operations last less than 100 microseconds, which is in the order of magnitude an iRMX-80 interrupt service requires. Therefore, the critical parts of the APU interface routines are protected simply by disabling the interrupt system, which evidently prevents other tasks from running and eventually accessing the APU. Since disabling and enabling the interrupts requires only one-byte machine instructions with an execution time of 800 nanoseconds each, this approach is clearly faster and more code-efficient than a software interlock; the synchronization between the CPU and the APU is done by a simple polling loop. Since the interrupts are never disabled for more than a couple of milliseconds, no significant deterioration of the interrupt response of the system is to be expected. (Interrupts happening at a higher rate than a few hundred per second are too fast anyhow to be reasonably processed by iRMX-80.)

Compared to the standard software floating-point algorithms of FORTRAN-80, the accuracy of the APU operations is slightly worse, due to the fact that the internal results are truncated to the mantissa length of 24 bits rather than being rounded. For most operations, the absolute values of the results obtained with the APU are therefore slightly less than those derived from software algorithms. The maximum relative differences lie in the order of 1.0E-6; in most cases, the relative differences between software and APU based results are less than 1.0E-7. For practical programming, this deterioration of the floating-point accuracy can be neglected.

5.2.5.2 Additional Routines in FP8231.LIB

In addition to the actual replacement routines for the standard FORTRAN-80 library floating-point algorithms, there are two functions available in FP8231.LIB, namely, ATANX and CABS.

ATANX: This function is equivalent to the FORTRAN ATAN2 function (inverse tangent of the quotient of two parameters). Still, this routine requires less execution time and less stack, due to its simpler internal structure. Two parameters of type REAL are required.

Call from FORTRAN:

```
result = ATANX (param1, param2)
```

with: result = arctan (param1/param2)
-PI/2 <= result <= PI/2

CABS: This function calculates the square root of the sum of the squares of its two parameters, corresponding to the absolute value of a complex number.

Call from FORTRAN:

```
result = CABS (param1, param2)
```

with: result = SQRT (param1**2 + param2**2)

5.2.5.3 The Implementation of the Alternative FORTRAN-80 Floating-Point Routines

The alternative FORTRAN-80 floating-point algorithms can be easily implemented in a system by including the library FP8231.LIB at system configuration time. With the exception of ATANX and CABS, the calls to the APU routines are inserted automatically by the FORTRAN-80 compiler; the replacement of the software floating-point algorithms is simply effected by linking the library modules in the proper order.

The current version of FP8231.LIB supports an iSBC 80-24 Single Board Computer with an iSBX 331 Multimodule Board installed in Multimodule connector J6 (base address 0F0H). Other base addresses can be used if a constant F@BASE holding the proper port address is declared PUBLIC in a module linked in front of FP8231.LIB. The programs using the alternative routines can be configured either as complete iRMX-80 systems, or as main programs to be run under RXISIS-II. RXISIS-II programs can be run without re-booting the system afterwards; no measures exceeding those required anyhow for the generation and execution of a real-time system are necessary.

The linkage sequence is in both cases:

```
RMX8xx.LIB (START) *)
Object Files
FORTRAN-iRMX-80 Interface Routines
FP8231.LIB
F80RUN.LIB
F80RMX.LIB or F80NIO.LIB
FPEF.LIB
FPSFTX.LIB
iRMX Libraries *)
PLM80.LIB
RXISIS.LIB #)

*) Not for RXISIS-II based programs.
#) Only for RXISIS-II based programs.
```

The stack requirements for the various operations are listed below. The two values given apply to an error-free and an erroneous operation, respectively. In the latter case, EH stands for the stack requirements of the error handler. The programs or tasks using the alternative FORTRAN-80 routines should provide the maximum stack required for a single operation, plus some reserve.

Addition:	26/48+EH
Subtraction:	26/48+EH
Multiplication:	26/48+EH
Division:	26/48+EH
Square:	16/50+EH
Square root:	26/56+EH
Common Logarithm:	46/88+EH
Natural Logarithm:	46/88+EH
Exponent:	46/88+EH
Sine, Cosine, Tangent:	26/58+EH
Inverse SIN, COS, TAN:	26/58+EH
ATAN2:	38/66+EH
ATANX:	32/60+EH
CABS:	20/52+EH
Conversion Bin./ASCII	34

5.3 The High-Level Growth Controller Software

5.3.1 The Operator Interface

5.3.1.1 The Console CRT Screen

The output on the CRT console terminal is the major visible part of the CGCS's Operator Interface. Several tasks some of which are not even part of the Operator Interface proper contribute to the console output (compare Fig. 9):

(1) Fixed Part (Lines 1 through 16 or 17):

Timer Task (FXTIME):
Actual and system time.

Command Interpreter Task (RXIROM):
Table frames, text output, date, and run identification.

Command Executor Task (CMMDEX):
Macro command name (if set); operation mode.

Measured Data Output Task (MEASDO):
All numeric values; Debug output in line 17 if activated.

Command File Input Task (CMFINP):
Macro command name (if cleared).

(2) Scrolled Part (Lines 17 or 18 through 21):

Command Interpreter Task (RXIROM):
Operator entry echoes, various messages.

Command Executor Task (CMMDEX):
Various messages.

Command File Input Task (CMFINP):
Various messages.

Diameter Controller Task (DIACNT):
Various messages.

All other tasks:
Disk, I/O, or system error messages.

(3) Prompt Line (Line 22):

Command Interpreter Task (RXIROM)

(4) Input Area (Lines 23 and 24):

Directly written to by the Terminal Handler.

The numeric values written to the console are, in general, given as physically relevant magnitudes, i.e., as properly scaled floating-point numbers. The following dimensions apply to the various items:

* Diameter, Lengths, Positions: Millimeters.
* Temperatures: Millivolts (thermocouple voltages).
* Lift Speeds: Millimeters per hour.
* Rotation Speeds: Revolutions per minute.
* Weights: Grams.
* Differential Weight: Grams per minute.
* Powers, Contact Device: Arbitrary units (0 ... 100).
* Gas Pressure: Pounds per square inch.
* Densities: Grams per cubic centimeter.

5.3.1.2 Auxiliary I/O Routines

The tasks which request console input or generate console output (compare chapter 5.3.1.1) use, in general, the FORTRAN-iRMX-80 Interface I/O routines whose names start with "FR..." to write to the screen, or the routines discussed in chapter 5.2.2.9 if they also write to the documentation file. All these output routines require a screen position information which is passed in the first parameter of the subroutine call. Some locations on the screen are, however, very frequently written to, and it was advantageous to provide special routines for these output actions which have the screen position information implicitly built in. Calling any of these "shorthand" routines spares the programmer entering one parameter, and it abbreviates the actual program code. Similarly, some input actions like the checking for the input string "Y(es)" can expediently be handled by dedicated routines.

The following routines (and several others) are kept in the assembly language module AUXASM. With the exception of PRETTA, they may be called by any task performing output.

PROMPT: This routine writes the string which was passed to it as a parameter left-adjusted into the input prompt line (line 22).

MESSGE: The string passed as a parameter to MESSGE is written into the scrolled screen area.

ERRMSG: Similar to MESSGE, the ERRMSG routine writes to the scrolled screen area, appending a "beep" in order to attract the operator's attention.

PRETTA: This routine writes "- press "RETURN" key to continue" to a specifiable screen location (usually in the prompt line), and waits for any input on the console.

Three additional I/O routines are kept in the FORTRAN module AUXCOM:

BEEP: This routine simply issues a "beep" on the system console. It takes no parameters.

CLIPRL: The subroutine CLIPRL overwrites the input prompt line with spaces. It does not take any parameters.

CHKANS: This routine is a LOGICAL Function. It returns ".TRUE." if a valid input line beginning with an upper- or lowercase "Y" was entered on the console, and otherwise ".FALSE.". CHKANS needs a LOGICAL argument which is returned ".TRUE." if an empty line ("Return" only) was entered, and otherwise ".FALSE.".

5.3.1.3 The Command Interpreter - Task RXIROM

The Command Interpreter task has a special position among the CGCS tasks in several regards:

* It is, in fact, the continuation of the ROM resident part of RXISIS-II, RXIROM, and the first task to come "alive" in the CGCS. Although it is "unofficially" referred to as "COMINT" within the program source modules, we will use here its "official" name RXIROM (which is also reported, e.g., by disk error messages).

* It performs the system initialization and activates all other CGCS tasks.

* It is the only task which requests and processes operator input (but not the only task to generate output).

5.3 The High-Level Growth Controller Software

The Czochralski Growth Control System is invoked under RXISIS-II by the command "CZOCHR". RXISIS-II searches for and loads a program module "CZOCHR.RXI" whose only purpose is to vector control to a special code sequence in the RXISIS-II Command Line Interpreter which replaces the file name extension ".RXI" by ".BIN", provides the resulting module name "CZOCHR.BIN" for the ROM resident bootstrap routine, and restarts the system. The bootstrap routine is part of the task RXIROM; normally, it loads into RAM and starts RXISIS-II. Being entered in the described way, however, it loads the module "CZOCHR.BIN" rather than "RXISIS.BIN" from disk drive 0; "CZOCHR.BIN" holds the entire resident code of the CGCS plus preliminary initialization values for some data locations, and a special start module which is loaded into the memory area which will later be used by the Command Interpreter overlays. Control is passed to this initialization code when the program file was successfully loaded.

The start module is entered via the assembly language routine CZINIT which first sets an internal flag of the Monitor which enforces a duplication of the Monitor's CRT output to the printer. (This measure provides a permanent printed record of an inadvertent entry into the Monitor program which might happen due to software or hardware failures.) CZINIT also resets a flag which controls the activation of the Monitor from the console keyboard. (This is why the Monitor can be entered under RXISIS-II but not from the CGCS by pressing the "Break" key of the console terminal.) Subsequently, CZINIT builds a new task stack close to the top of memory since the stack of RXIROM is too small. It stores a program version code in a reserved memory location; later, a version code which is loaded with each overlay will be compared to this datum in order to ascertain that only matching program modules are loaded. After some initialization calls to FORTRAN and FORTRAN-iRMX-80 Interface routines, CZINIT passes control to the FORTRAN subroutine FXUSIN.

FXUSIN initializes the digital I/O interface and several control structures which can be accessed more conveniently via FORTRAN than via assembly language. It calls the (assembly language) subroutine TESTHD which checks whether an A/D converter board is installed in the system by initiating a conversion and checking the status byte for a "Conversion Ready" bit which is returned by the A/D converter. The Variable TEST is set to -1 if no A/D converter response was detected within a defined timeout period; otherwise, TEST is returned with the value 0. (This check is important if the CGCS software might be run on hardware which does not feature the A/D and D/A interfaces. In this case, practically all system resources would be spent by the task ANACNT for waiting for the A/D con-

verter to finish a conversion, which obviously never happens if there is no A/D converter within the system. The CGCS would, therefore, be practically locked in such a test environment. The value of TEST is later used for bypassing the analog input and output routines within the task ANACNT. Note that TESTHD is called before ANACNT is created.) Subsequently, FXUSIN calls the assembly language subroutine CREATE which is, similar to TESTHD, part of the module CZINIT. CREATE activates all tasks of the CGCS, which can only be done safely after the above initialization, and makes unused memory (including the old RXIROM stack) available to the memory pool of the iRMX-80 Free Space Manager. After the return from CREATE, FXUSIN provides a sign-on message (plus a message referring to a "Test Mode" if TEST has been set to -1), and loads the data overlay "CZOOVD" from drive 0.

Similar to "CZOCHR.BIN", "CZOOVD" is loaded only once during every growth run. "CZOOVD", which holds the (initialization) values of practically all system parameters, is kept separate from the main code module on purpose. The preparation of the CGCS program modules is a lengthy and complicated procedure which would have been indispensable after each modification of a system parameter initialization value if these data had been kept within "CZOCHR.BIN". Since it is very likely that numeric parameters require changes more frequently than the program code, it was preferable to load them from a special data overlay which can be modified and configured relatively easily.

The auxiliary routine LOVLAY which is exclusively used by RXIROM loads overlay modules into RAM. (The information where an overlay is to be loaded is part of the overlay program file. It is, therefore, sufficient to specify the name of the file to be loaded.) Several safeguards are provided which permit to trap the potentially disastrous loading of improper files:

* The data on each disk file and, in addition, the program code itself, contain checksums which are validated by the Loader task. Any damage to a program file is therefore very likely to be detected and reported by the Loader. LOVLAY returns a message "Defective program disk" in this case.

* Each overlay contains memory locations which hold its name and the program version code. LOVLAY reports "Software damage likely - reset the system" if either the overlay name or the program version loaded with the overlay do not match the expected data. (It is important not to mix modules belonging to different CGCS versions because all overlays access code or data within the resident part of

the CGCS. Since the absolute address of a routine or a data location may change due to system modifications, an overlay routine may call improper code or access wrong data if its version does not correspond to the version of the resident code.)

Note: Do not disregard error messages returned during overlay loading. A potentially disastrous effect of a defective overlay may show only after a considerable time. It is always dangerous to copy single overlay files to a work disk, or to exchange work disks inconsiderately. There is, however, no danger if a Disk Error 24 is reported during overlay loading, and if the defective disk is replaced by one which holds the same program version.

In very rare cases, a Disk Error 120 - Unable to Open File - may be displayed when the system attempts to load CZOOVD. This may happen if the operating system is overburdened during the start phase, for example, if a key is continuously being pressed on the console terminal. In this case, the memory pool has not been initialized yet when memory is requested from it by the Loader software, and the above error condition ensues. The "Defective program disk" message may be ignored in this case, and loading may be retried by pressing "Return".

The start routine FXUSIN displays the creation date of the data overlay CZOOVD (which is also an indication that this module was loaded properly), and requests the current date. The date should be entered in the format shown in the prompt, but any string of 8 characters which starts with a digit is accepted. The date information is stored for reference purposes only; it will be used on the console screen, in the documentation output page headers, and in the header records of the Data files. After the date, the current time is requested from the operator; the system expects two or three positive integer values as an input, separated by colons (":"), spaces, or any other non-numeric characters. The time should be entered in 24 hours format; zero is assumed as a seconds value if only hours and minutes were specified. The internal system time starts running - yet invisibly - when the subroutine FRSETT is called after the "Return" key was pressed to enter the time information, and the absolute time is set to the value entered (compare chapter 5.2.4.1). Finally, FXUSIN requests a "Run Identification" which can be any arbitrary string up to 20 characters long. A blank run ID can be entered by simply pressing "Return".

FXUSIN calls now the subroutine TIMLIN which is part of the start code in the future overlay area. TIMLIN generates the date, absolute time, run ID, and system time display in the

top screen line which will be shown throughout the entire growth run. The operator can accept or reject the data displayed; this is, by the way, the only occasion within the entire CGCS software where a plain "Return" is interpreted as "Yes" (otherwise, it is treated as "No"). Depending on the outcome of this query, FXUSIN either loops again through the date, time, and run ID input section, or it returns to CZINIT which passes control to the resident portion of the Command Interpreter, i.e., to the routine COMINT.

COMINT starts its operation by writing the output "frame" to the console terminal which is eventually filled in with the output of measured data. This is done by the subroutine FRAME which resides in the overlay CZOV08. This overlay has to be loaded by COMINT; it overwrites the code of CZINIT and FXUSIN. (This and the following initialization can, therefore, not be done from FXUSIN which would otherwise be the logical place to do them; there is no way for a routine in an overlay to call directly another overlay resident routine which uses the same physical memory locations.) We will discuss the subroutine FRAME later which is also called upon a RESTORE command.

The next two routines invoked during the initialization of COMINT reside in overlays CZOV16 and CZOV19, respectively. DOCUMT permits to activate a Documentation output, either on the printer or on a disk file, and it allows to specify an interval for dumps of measured data to the Documentation output. INIDAT permits the initialization of some process parameters. Both routines will be dealt with later.

COMINT enters now its infinite loop which starts with the output of the prompt "Please command:" and the request of operator input. The input routines transfer an input line of up to 80 characters into an internal buffer when the operator terminates his entry with "Return"; no data are available to the CGCS before "Return" is pressed. First, COMINT attempts to transfer the first six characters in this input buffer to the CHARACTER variable COMMD. The LOGICAL variable STAT is returned ".TRUE." by the STRIN call if and only if an empty line was entered ("Return" only). COMINT repeats its input prompt in this case, and waits for the next entry. Otherwise, the first character of the input line is checked and the input rejected if it is a space (the command keywords must be left adjusted within the input line to be processed properly).

The presumable command keyword in COMMD is now compared to (currently) 25 keyword strings, corresponding to the 24 Internal commands (the HELP command has the alternate keyword "?"). Control is vectored to the appropriate sequence within COMINT if a matching string is found. (The string comparison routine

FRCMPS uses the character "|" as a wild card symbol which can be matched to any character (compare chapter 5.2.4.3); only four characters are compared since the keyword strings consist of four or less characters only.) The command entry is interpreted as the name of a Macro command if no matching Internal command was detected. A Macro file name string is created by the assembly language routine MAKEFN which appends the file name extension ".CMD" at the logical end of the presumptive Macro command name (which is either after the sixth character of COMMD, or at the first space in COMMD, whatever happens first). COMINT tries to open the Macro file for reading, and closes it immediately, in order to test whether a file with the specified name and the extension ".CMD" does exist. This is the case if no error status is returned by the FROPEN call; FROPEN will return an error value of 13 ("No such file") if no Macro file was found with the specified name, most likely due to a mistyped command. An error value of 4 ("Illegal file name") may be returned if the command input contains non-alphanumeric characters, which may also happen due to typing errors. COMINT returns to the beginning of its command loop with an appropriate message in these cases; the default disk error message is output if any other disk error was detected. If a file with the proper name was found, COMINT assumes that it <u>is</u> a valid Macro file (this fact will be checked later); it requests an operator acknowledgement ("Execute Macro command ...?"), and sends a command message to the Command Executor which eventually will start the execution of the Macro command.

In general, all commands which may be recorded on and issued by a Macro command file are executed by the Command Executor. These commands are "sent" to the Command Executor by means of messages, buffer areas in RAM which are made available to the receiving task by the iRMX-80 operating system. Command messages have a "type" value of 161 (the message "type" is simply a safety feature which guarantees that correct data is received). The first byte of the command message proper determines the command type (in our case, 30H stands for "Macro Command"), and the remainder of the message holds parameters of the specific command, up to a length of 13 bytes. The same format, with two additional leader bytes holding the command time, is used to store commands within a Macro file; compare Appendix 12. Using special message transmission routines of the FORTRAN-iRMX-80 Interface Program Package permits to easily merge command messages from different sources (namely, from the Command Interpreter and the Command File Input tasks) and to queue them at the Command Executor's input for processing. After having been accepted by the Command Executor, the command messages are passed on to

the Command File Output task which records them in a Command Output file (Fig. 17).

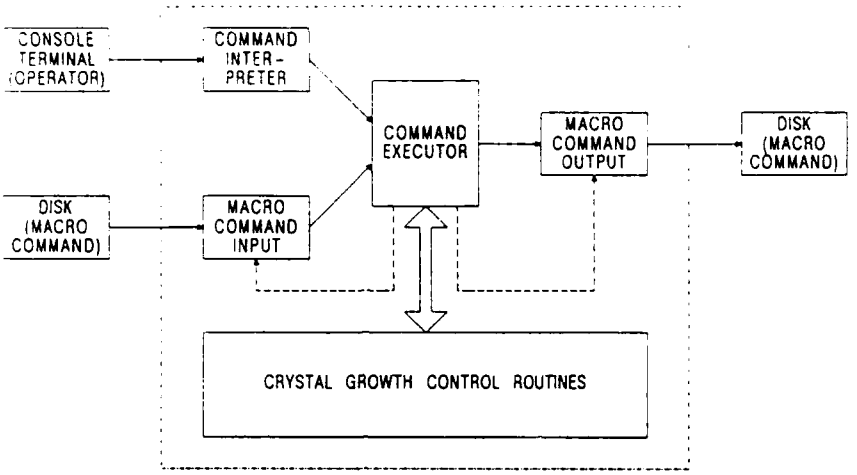


Fig. 17: Command processing in the CGCS.

Most of the Internal commands are processed in overlay resident routines which we will discuss later, rather than within the main Command Interpreter routine COMINT. This approach helped to keep the resident COMINT code concise. Only the following commands do not require overlays to be loaded:

EXCHANGE: This is considered an emergency routine which must be called if a disk has to be changed due to any kind of defect. It would not make sense to load an overlay from a possibly defective disk. In order to prccess the EXCHANGE command, COMINT calls the FORTRAN subroutine XCHDSK which closes all files on the specified disk, waits for an operator entry which indicates that the disk has been exchanged, and re-opens all output files on the new disk.

END: An End of Command Record code (7FH) is sent to the Command Executor if this command was issued.

QUIT: The FORTRAN subroutine QUITCM which is invoked by COMINT disables the Macro command file input (by resetting the proper I/O flag) and the Timer #2 which controls the

execution of Macro commands. It fakes a timer alarm by setting the flag TIMINT, and waits for two iRMX-80 time units (100 ms) to permit the Command File Input task CMFINP to run in response to the faked alarm. CMFINP closes the Macro command file, clears the Macro name on the top line of the console screen, and issues a corresponding message if it finds the I/O flag reset.

DUMP: The subroutine DUMP which is called immediately upon a DUMP command sets a flag (DUMPFL) to .TRUE. whose status is periodically checked by the Command Executor (compare chapter 5.3.1.4.6). The Command Executor, in turn, initiates a Data Dump to the Documentation output when it finds this flag set.

All other commands are handled by the overlay resident routines. In order to avoid loading an overlay which has already been loaded by a preceding command, COMINT checks the value of the variable OVRLAY which is set by each overlay to its respective overlay number. (This is not explicitly done by program code but by assigning a value to OVRLAY with a BLOCKDATA program; this value is stored in OVRLAY when the overlay is loaded.) The COMINT overlays are discussed in the following chapters in their numerical order which has been determined essentially by historical reasons.

5.3.1.3.1 Overlay CZOV01 - Module SETPAR - Commands SET and CHANGE

The subroutine SETPAR receives the MODE switch as a parameter which distinguishes between the SET and CHANGE commands, and it returns the LOGICAL variable LOAD. LOAD is returned ".FALSE." if SETPAR can complete the processing of the command, i.e., if the command applies to one of the nine Internal parameters (diameter, three temperatures, four motor speeds, and power limit). Otherwise, LOAD is ".TRUE.", and COMINT has to load the overlay CZOV02 in order to complete command processing.

SETPAR re-scans the command line originally issued to COMINT, searches in it for the first space, and then for the first three alphabetic characters after the space, in order to determine the parameter which is to be SET or CHANGEd. An explicit request for a parameter is issued if no suitable data are found in the input line, and a new input line is read and parsed for its first three characters. The command is cancelled if this second attempt is also unsuccessful. In either case, the input line pointer is moved back to the beginning of

the parameter string, i.e., the next input command will read the parameter string again unless a search option is used with the input routine call. The three input characters are now compared to the nine mnemonics which stand for the primary parameters (three characters are required because the third of them must be a space in order to match a valid mnemonic). SETPAR is left immediately, with LOAD set to ".TRUE.", if no matching mnemonic is found.

The routine scans now to the first space after the parameter string and tries to read a valid floating-point number from the input buffer. This number will represent the target value of a SET command or the increment of a CHANGE command. A proper value is requested if no numeric value is found after the parameter string, and a new input line is read in this case. Either input line is scanned for the next (floating-point) number, and a transition time entry is prompted for if no such number or a negative value is found. The command is regarded cancelled if no valid input is entered after it was explicitly requested. A similar approach is used within all Command Interpreter routines which process commands permitting the entry of command parameters in the input line.

In order to generate an operator confirmation prompt, SETPAR determines the final value of the modified parameter. This value is equal to the input value for a SET command but must be calculated as the sum of the current parameter value and the specified increment in the case of a CHANGE command. Internally, the setpoint and actual values of the primary parameters are stored as scaled two-byte integer (INTEGER*2) values. This was done because analog data are input and output as integer values; the controller routines operate on integers because integer algorithms are faster and require less code, and data recorded in the Data file are also in integer format, which reduces the Data file size by a factor of two, compared to floating-point numbers. The physically relevant (floating-point) data which are displayed and entered on the console are obtained from the internal integer values by multiplying them with appropriate scaling factors.

One peculiar property of real-time systems must be considered at this point: Unlike conventional computer programs, routines which are part of a real-time system may not freely read and write data. This is true because multi-byte values are usually stored and retrieved in sequences of several machine-code instructions. The scheduling of system tasks is, however, hardly predictable in a real-time environment, and a task might be interrupted, e.g., during a multi-byte read, by another task which might write to the same memory locations. Although the actual value stored in these memory locations

might change only slightly, a totally unusable value might be retrieved by the interrupted task. Such an event may be relatively unlikely but nevertheless disastrous; the following safety measures are taken within the CGCS to prevent it (compare also chapter 3.1.4):

(1) Some data areas are protected by access control routines (FRACCS and FRRELS) which permit only one task at a time to read them or write to them.

(2) The system Variables are implicitly protected by the proper choice of the priorities of tasks which access them. They are only written to by the Command Executor which has a very low priority and can therefore never interrupt the execution of a higher-priority task which might use a Variable. The storage of the Variables is protected by using a special routine (STODAT) which temporarily disables the system interrupts.

(3) Values which have to be read only can be retrieved reliably by reading them twice. This process can be repeated until both reads result in the same value.

The latter approach is the one chosen in SETPAR; a counter prevents the system from being blocked in the unlikely case that a matching value pair is never found.

SETPAR checks the final setpoint for negative temperature or power limit values, and requests an operator acknowledgement. The output line has, unfortunately, to be built relatively awkwardly in a buffer (LINBUF) because the output routines which write also to the Documentation file can only accept a complete line of output (compare chapter 5.2.2.9).

Upon a positive answer of the operator, SETPAR builds the command message. The command type byte holds the encoded command mode (SET or CHANGE) and the target parameter; the input value is converted to an INTEGER*2 (which is checked for a potential overflow), and the transition time value which was specified in minutes is multiplied by 60 to hold a ramping time in seconds. The command message is dispatched to the Command Executor, and SETPAR returns to the resident COMINT code.

5.3.1.3.2 Overlay CZOV02 - Module SETVAR - Commands SET and CHANGE

SETVAR is invoked after a SET or CHANGE command for which none of the Internal parameters was specified. The CGCS assumes in this case that the command applies to a system Variable, i.e., to an item in a list of named memory locations. SETVAR receives the input buffer from SETPAR with the pointer at the first character of the presumptive Variable name; it reads a string of up to 10 characters into an internal buffer, terminating the input action when a space (i.e., the end of the Variable name) is encountered. The name string is converted to uppercase, and passed to the assembly language routine FINDAD.

FINDAD compares the presumptive Variable name in VARNAM to a list of names kept in the specially formatted file CZONAM.Vmn, where m and n are the major and minor program version numbers, respectively. Each entry in this file holds a Variable name (1 to 6 alphanumeric characters long, but the first character must be alphabetic), the Variable type (one- and two-byte integers or four-byte floating-point numbers), encoded with the number of elements if the Variable name refers to an array, and the Variable address or the start address of an array (compare Appendix 12). FINDAD checks the index of an array element which may optionally be passed in parentheses immediately after the Variable name, and returns the actual address of the Variable or array element, and a type code which is positive if a valid entry was found in the CZONAM file, and negative in case of an error.

SETVAR checks the type code returned and issues an error message if necessary; otherwise, it retrieves the current value of the Variable. This is done with a call to the assembly language subroutine PEEKDW which reads the four bytes at the address passed as a parameter repeatedly until a stable result is obtained (compare chapter 5.3.1.3.1). The four bytes read may have to be converted to a floating-point number according to the type of the Variable; the result of this operation is later used to display the current and the final values of the Variable. Subsequently, the routine tries to obtain a SET or CHANGE final value and a transition time from the input buffer, and it issues corresponding prompts if no data are found.

Similar to SETPAR, SETVAR checks integer values for a valid range, builds a command message if the operator acknowledgement was positive, dispatches the message, and returns to COMINT.

5.3.1.3.3 Overlay CZOV03 - Module COMMEN - Command COMMENT

The routine COMMEN inserts a comment line into the Data file if such a file is active.

COMMEN scans to the first space in the original command input line, and tries to read valid input from the remainder of the command line (to receive any comment which was entered together with the COMMENT command). A corresponding prompt is issued if the command line did not contain any data except the keyword. COMMEN returns immediately to COMINT if no Data file is active (i.e., IOFLAG(2) is reset); otherwise, it provides operation mode, time, and length grown information in its output buffer, sets the first byte of this 128 byte buffer to -1 to indicate a comment line, and writes the buffer to the Data file. It is essential that a full record (128 bytes) is appended to the Data file to maintain the file's special format (compare Appendix 12).

5.3.1.3.4 Overlay CZOV04 - Modules MENOUT and CLRSCR - Command HELP

This overlay provides the Help menus of the CGCS in response to the commands HELP and "?". It writes in random access mode into lines 17 through 21 which are otherwise reserved for scrolled and Debug output. The latter is immediately disabled when MENOUT is entered by resetting the flag ENDBGO. Although the output routines do permit to write over the scrolled area in random access mode, this output remains on the screen only until data is output again in scrolled mode. Any system message which is issued while the HELP command is executed will therefore preempt the display of the current Help menu.

MENOUT first clears the five lines of the scrolled area by overwriting them with spaces (in the subroutine CLRSCR), and outputs a quick menu of Internal commands which is built right into the program. The next help screen optionally displayed by MENOUT contains a list of Macro command names which are derived from the disk directory of the disk in drive 0 (file ISIS.DIR). The directory is scanned for all valid files with an extension ".CMD". Up to 40 Macro files can be listed on one screen; if there are more Macro files on the system disk, MENOUT pauses and continues its output when the operator pressed the Return key.

After displaying the Macro commands, MENOUT permits to request more information about the Internal commands. If the operator accepts this offer, MENOUT displays again the short menu.

(The initial menu display sequence is also used for this purpose; a LOGICAL variable controls the continuation of the execution of MENOUT after the menu was output. This approach was chosen rather than a subroutine call because it is more program code efficient, and because it does not require awkward measures like COMMON blocks or lengthy subroutine parameter lists to make variables available to all routines involved.) Simultaneously, MENOUT opens the help file CZOMEN for reading which holds five lines of text for each command. There are two modes in which the contents of CZOMEN can be displayed: One mode steps through the file, displaying record by record, while the other one scans the file until a keyword entered by the operator is found in the first line of an entry; only this entry is displayed. Both modes can be combined since an empty input line ("Return" only) always results in the next record being displayed, whereas the first four characters of a non-empty input line are used to search through the file CZOMEN. Multiple entries can therefore be searched for in one pass, provided they are in ascending alphabetical order. A single-character entry (nominally, "Q", but any other character has the same effect) terminates the search, and MENOUT is exited after closing the menu file and re-enabling a possible Debug output by setting the flag ENDBGO.

5.3.1.3.5 Overlay CZOV05 - Modules OPMODE and CLRSCR - Command MODE

The operation mode setting routine OPMODE displays a mode menu similar to MENOUT, and permits the entry of a mode number. The number entered is compared to the current mode and checked for its validity; corresponding messages are output if either the current mode was chosen, or if an illegal mode number was entered. OPMODE permits to re-select the current mode; although this has no effect whatsoever on the current growth run, the command is recorded in the Command Output file and may be effective during a later execution of this file as a Macro command. (It may also be used to trigger a data dump on the printer and in the Data file; there are more straightforward methods to achieve this, though.)

The operator is prompted for an acknowledgement of his mode entry in any case. OPMODE requests an extra acknowledgement (with "OK" rather than "Y(es)") if the mode is changed from Monitoring (mode 0) to any controlled mode, or vice versa, in order to prevent the probably disastrous effects which an inadvertent change might have. The newly entered mode is encoded in a command number, and the command message is sent to the Command Executor.

5.3.1.3.6 Overlay CZOV06 - Module DEBUG0 - DEBUG Commands

The six DEBUG sub-commands - Continuously, Display, Modify, Off, Resume, and Suspend - are handled by the two overlays CZOV06 and CZOV07 (modules DEBUG0 and DEBUG1, respectively) which are concatenated similar to the two overlays for the SET and CHANGE commands. The command execution is commenced in the module DEBUG0 where the command input line is first scanned for the DEBUG mode switch, which is any one of the letters C, D, M, O, R, and S. As usual, a mode switch is requested if none or only an illegal one was found.

The processing of the DEBUG commands requires various interpretations of the input line, depending on which sub-command was issued. In order to facilitate this processing, the entire contents of the input buffer are read into an internal buffer (LINBUF) from which input items are retrieved. The contents of this buffer are shifted to the left by one item after each successful input operation, which permits to read the next item always from the beginning of the buffer. (Items must be separated by spaces; the buffer shifting subroutine SHIFTB simply advances to the first non-blank character after the first space and copies the buffer onto itself from this location on.)

For all sub-commands except Off, either the name of a Variable or an address is required as the first parameter. An input item starting with a number is considered a (hexadecimal) address, otherwise, the parameter is submitted to the routine FINDAD which was already discussed in chapter 5.3.1.3.2. The DEBUG routines distinguish between address and Variable input by setting the Variable type location VARTYP to -1 in the case of address specification, whereas values from 0 to 3 are returned by FINDAD for Variables.

Indeed, the information otherwise provided by FINDAD in the Variable type location must be obtained from the operator if address input was chosen since DEBUG would not know how to interpret the data at the specified address. (This information is <u>not</u> needed for the Display sub-command which outputs data anyhow in all perceivable notations.) A data format is, therefore, retrieved from the input buffer or requested from the operator if an address value was specified with a Continuously or Modify sub-command. (The formats used for numeric Variables are internally set to "I1", "I2", and "R", depending on VARTYP.)

The Continuously and Off sub-commands require a Debug Channel number, i.e., the number of the output location in the Debug line (1 to 4) which the command refers to. For both sub-com-

mands, all necessary information is now available, and the proper command messages can be sent to the Command Executor.

The Display and Modify sub-commands display the current contents of the specified memory locations; in order to obtain this datum, four bytes beginning with the given address are copied into local memory in an approach similar to the one used in SETPAR and SETVAR (compare chapter 5.3.1.3.1). The contents of the specified location(s) are is immediately displayed in several modes if the Display sub-command was issued: The four bytes or part of them are interpreted as ASCII string data, as an INTEGER*1 and INTEGER*2 variable, as floating-point data (type REAL), and as hexadecimal numbers. (A special treatment is necessary for the ASCII interpretation in order to avoid problems with data bytes which might be interpreted as control codes by the console terminal. Such bytes are replaced by periods (".").)

While the Continuously, Display, and Off sub-commands already have been completely processed when the end of the module DEBUG0 is reached, this is not true for the Modify, Resume, and Suspend commands. They have to be passed on to the second part of the DEBUG routine, DEBUG1 in CZOV07.

5.3.1.3.7 Overlay CZOV07 - Module DEBUG1 - DEBUG Commands

Similar to SETVAR, DEBUG1 is only loaded if DEBUG0 returns with a status flag set to ".TRUE.". Data are passed between both routines by means of a special named COMMON block (DBG-COM) which is located at the top of the overlay area where it is preserved when DEBUG1 is loaded.

In order to conclude the processing of the Modify sub-command, DEBUG1 displays the current contents of the specified memory locations, and requests explicitly new data. Both values are displayed again for operator confirmation, and the command message is built if the confirmation was given.

The sub-commands Resume and Suspend which permit to resume and suspend the execution of an arbitrary task are treated essentially in common: Both require the address or the name of an iRMX-80 Task Descriptor as a parameter. Task descriptors which are referred to as Variables have the Variable type value of zero returned by FINDAD. Since specifying an address with a Resume or Suspend system call which is <u>not</u> the address of an iRMX-80 Task Descriptor would have a disastrous effect on the entire system, multiple safeguards are used besides checking the VARTYP value: The name of the task, six alpha-

numeric characters, is stored in memory locations whose start address can be derived from the presumptive Task Descriptor. The command is cancelled if either non-alphanumeric characters are detected in the name area, or if the first character is not alphabetic. After an operator acknowledgement, a proper command message is again dispatched to the Command Executor which will, in turn, resume or suspend the specified task.

5.3.1.3.8 Overlay CZOV08 - Modules FRAME and TIMLIN - Command RESTORE

This overlay provides the mask for the "fixed" output on the console CRT screen. It is executed upon a RESTORE command, and during the system initialization.

FRAME which is in charge of the main output mask first disables the output of measured data by resetting the flag RESTDO(3). This is important to avoid interferences between the two groups of output operations. Furthermore, Debug output in line 17 is suspended by resetting the flag ENDBGO. FRAME clears the CRT screen, and calls TIMLIN which restores the top (time) line. Subsequently, all fixed output items are written one by one, followed by a five line parameter dimension information written over the scrolled screen area. Having provided this menu, FRAME enables the output of measured data by setting RESTDO(3), and actually enforces data output by setting the remaining two flags of the array RESTDO. FRAME pauses then until the operator presses the "Return" key to permit him to read the display in the scrolled area. Writing a blank line into the actual scrolled output restores the previous contents of the scrolled area after a RESTORE command.

5.3.1.3.9 Overlay CZOV09 - Module FILES - Command FILES

The subroutine FILES permits to display the current status of the three output disk files (the Documentation, Data, and Command file), and to change the status of a selected file.

First, FILES displays the name and the status of each file. The file names are kept in the CHARACTER array FILNAM; the file status is determined by the values of IOFLAG and FILLOC. The proper element of the array IOFLAG is set to ".TRUE." whenever a file is actually active, i.e., data can be written to it. FILLOC, in contrast, represents the physical location of a file; 0 and 1 stand for drive 0 and 1, respectively, and

2, for output to the printer. FILLOC is set to 3 if no file is open at all. In the case of the Control Output file, the setting of the flag RECORD has also to be taken into account which is ".TRUE." while commands are actually recorded (i.e., after a START command), and ".FALSE." otherwise.

The operator may now specify one of the three output files which he wants to be opened and closed, or return immediately to COMINT. The actual file treatment is performed by one of three separate overlays; the proper overlay number is determined by FILES and passed to COMINT in OVRLAY; COMINT concatenates the proper routine.

5.3.1.3.10 Overlay CZOV10 - Module REQCMF - Commands START and FILES

The subroutine REQCMF can only be called via the START and FILES commands; it opens, initializes, and closes Control (or "Command") Output files.

The response of REQCMF depends on the status of the file; it distinguishes between three cases:

(1) No Command Output file is open (IOFLAG(3) is ".FALSE.", and FILLOC(3) is 3).

(2) The file has been opened, but it cannot be written to due to a preceding disk error (IOFLAG(3) is ".FALSE." but FILLOC(3) is not equal to 3).

(3) The file is open and active (IOFLAG(3) is ".TRUE.").

In the first case, REQCMF offers the operator to open a Control Output file, and requests a file name if he agrees. A complete Macro file name is built from the operator's entry by appending ".CMD" (with the subroutine MAKEFN), and the resulting file name is checked for validity and for the drive where the file will be located (with CHKFNM). In order to prevent the accidental overwriting of an existing file (if the operator entered the name of a file which already exists on the same disk), REQCMF tries to open the file with the specified name for reading first, and issues a warning if this procedure was successful, i.e., if a matching file was found. Otherwise, the Command Output file is opened for writing, and a header record is written to it. The header record holds zeros in its first two bytes (which otherwise contain the execution time of a command), and the system version code in the third and fourth byte. The remaining 12 of the 16 bytes of the

header record are currently undefined. REQCMF finally sets IOFLAG and returns to COMINT.

In the second case, REQCMF permits either to re-activate the file (possibly, the error condition has already been corrected which set it inactive), or to close it. An open and active file may be closed only; if the operator agrees to close the file, IOFLAG and RECORD are reset, FILLOC is set to 3, and the file name string is deleted.

5.3.1.3.11 Overlay CZOV11 - Module CALCUL - Command CALCULATE

The CALCULATE command constitutes a helpful utility which is, in fact, not connected to the crystal growth process at all. CALCUL permits to evaluate the sum, the difference, the product, and the quotient of two numbers. With regard to the requirements of the DEBUG commands, three formats are selectable for input and output data, namely, (two byte) Integer, Hexadecimal, and Real (floating-point). One set of instructions applies to the processing of floating-point input values, and an other, to integer and hexadecimal data. The results are displayed in decimal and hexadecimal notation in either case; the CALCULATE command may therefore be used to determine the internal (hexadecimal) representation of an arbitrary integer or floating-point value.

5.3.1.3.12 Overlay CZOV12 - Module DATAFI - Commands FILES and DATA

With the exception of the header record generation, the routine DATAFI which is responsible for the initialization and maintenance of the Data file is analogously identical to REQCMF (compare chapter 5.3.1.3.10).

A data sampling interval (in seconds) is requested from the operator when a Data file is opened; any value between 1 and 255 is accepted. The header record is built after the operator acknowledged the interval value. It contains the date, the run ID, the data records interval, and the system version. This header which is 32 bytes long is written to the newly opened Data file four times, to permit the first Data record to start at a disk sector boundary. (This is important because disk operations are much faster if an entire disk sector can be written or read.)

5.3.1.3.13 Overlay CZOV13 - Module EXICZO - Command EXIT

This module has the chore of "closing down" the CGCS and the puller. It requires a double acknowledgement by the operator to be actually executed, in order to prevent accidental exiting from the CGCS. It performs the following operations:

(1) EXICZO disables periodic data dumps to the Documentation output.

(2) It sends an END command to the Command Output file if such a file is still open.

(3) It clears potentially pending Conditional Commands by transmitting a CLEAR command code to the Command Executor.

(4) It performs a QUIT command (calling QUITCM) to preempt a currently active Macro.

(5) It switches off Data recording by de-activating a Data file (setting IOFLAG(2) to ".FALSE.").

(6) It shuts the system down with the following actions if the digital system is actually controlling the puller:

    (a) It terminates automatic growth, changing the operation mode to "Manual" by sending an appropriate command message.

    (b) It terminates any parameter ramping possibly still in progress by resetting the Ramping flags RMPFLG (compare chapter 5.3.1.4).

    (c) It checks the current values of the motor speed and power limit setpoints, and enters into the following actions if any one of them is not equal to zero:

        (1) It permits the operator to skip from EXICZO and to shut the system down on his own account.

        (2) It ramps the power limit setpoint to zero within approximately 6 hours unless it is already zero, generating an appropriate command message.

        (3) It ramps the seed and crucible lift speeds to zero within one minute.

        (4) It provides a time countdown in the input prompt line which starts at 360 minutes if the power

limit need be ramped down, and otherwise at one minute.

(5) It ramps the seed and crucible rotation speeds to zero within one minute when the countdown display shows one minute.

(d) It prompts the operator to switch off the puller's power supply, and submits control to the analog controller when the operator indicates that this is possible by commanding "EXIT" again. The operation mode is set to Monitoring with a suitable command message.

(7) EXICZO disables the output of measured data, and stops the Measured Data Output Task (compare chapter 5.3.1.5). Simultaneously, it resets the Timer Output Enable flags ENTIMO to prevent the display of new time strings.

(8) It closes all output files which are possibly still open,

(9) Clears the console screen and writes a sign-off message,

(10) Switches all output relays off, and

(11) Calls the routine FREXIT which will re-boot RXISIS-II.

5.3.1.3.14 Overlay CZOV14 - Module CONDIT - Command IF

CONDIT receives the command input line buffer from COMINT; it tries to retrieve a Variable name from it by scanning to the first non-blank character after the first space. A Variable name is requested if none was found. This name is converted to uppercase (with FRCVUC) and processed by FINDAD which returns the address and the type of the Variable specified. Next, one or two relational characters ("<", "=", or ">") are either read from the input buffer, or explicitly requested. A numeric value of 1 to 3 is assigned to each valid relational character; the two relational characters and the Variable type are packed into one byte of the command message in order to conserve space. After a comparison value which is simply stored in the command message, the name of the Macro command which is to be executed conditionally is retrieved in the standard way. A Macro file name is built from the command name (with MAKEFN), and CONDIT tests the requested Macro file in the same way which COMINT uses for the same purpose. The Macro name is stored in the command message which is dispatched if the file exists and the operator acknowledgement was obtained.

5.3.1.3.15 Overlay CZOV15 - Module DISPLY - Command DISPLAY

DISPLY requires the name of the Variable whose value is to be displayed as its only input. The name is either read from the input line buffer, or explicitly requested. After a conversion to uppercase, it is submitted to FINDAD which returns the address and the type of the Variable. The Variable is read subsequently with the algorithms already discussed above (compare chapters 5.3.1.3.1 and 5.3.1.3.2), and displayed according to its type.

5.3.1.3.16 Overlay CZOV16 - Module DOCUMT - Commands FILES and DOCUMENTATION

The module DOCUMT is accessed during the initialization sequence, from FILES, and at a DOCUMENTATION command call. DOCUMT is very similar to REQCMF (compare chapter 5.3.1.3.10) and to DATAFI (compare chapter 5.3.1.3.12). The major differences between these routines are (aside from the different IOFLAG and FILLOC array elements which they use):

(1) DOCUMT explicitly permits to use the printer as an output device (which would not make sense with the other two files).

(2) It permits to specify an interval for the periodic output of measured data to the Documentation file, and

(3) It opens the Documentation file, enables printer output (in case it was disabled due to a printer timeout), and initializes the output routines with a call to the routine STARTP which is part of the DATOUT module (compare chapter 5.2.2.9). STARTP presets the line counter and generates a page header line in the Documentation file.

5.3.1.3.17 Overlay CZOV17 - Module DIRECT - Command DIR

DIRECT displays the directory of the disk in the drive specified, together with some information about the disk itself. Having obtained a valid drive number (0 or 1), DIRECT first reads the disk label, i.e., the name of the disk, which is kept in the file ISIS.LAB. Next, the routine determines by checking the file location array FILLOC whether files are open for output on the specified disk. In this case, the information about the free and used disk space cannot be reliably obtained (because iRMX-80 maintains a disk allocation map in

its internal memory and writes it back to the disk only when a file has been closed), and the corresponding values are preceded by "less than" and "greater than" signs, respectively. The number of occupied sectors on the disk is retrieved from the disk map file ISIS.MAP; each bit set in ISIS.MAP corresponds to a used sector. These bits are counted by the (assembly language) Function BITCNT, and written to the directory header line. Since each single-density, single-sided 8" disk holds 2002 sectors of 128 bytes each, the number of free sectors can be calculated easily. Finally, DIRECT reads the disk directory file ISIS.DIR, and displays all file names in a way similar to the one chosen for the Help menu output (compare chapter 5.3.1.3.4). DIRECT can only output 6 entries per screen line since full file names, including extensions, have to be displayed. The routine pauses after having written the header line and four lines of directory contents, and continues overwriting the directory display with new data after the operator pressed the "Enter" key. DIRECT returns to COMINT when the "Enter" key was pressed after the last valid directory entry was displayed.

5.3.1.3.18 Overlay CZOV18 - Module RESOVL - Command RESET

The RESET command is indispensable for the initialization of the diameter evaluation routines. It prepares not only the buoyancy compensation routines in the module SHAPE (compare chapter 5.3.2.2) but initializes also the weight and length values displayed. RESOVL offers the standard option of resetting length and weight to zero; it permits to maintain the current value for each of these parameters or to enter new values if the zeroing option was rejected. The values input by the operator are scaled to obtain integer data in the formats used internally for weight and length representation; a value of -32768 (the most negative integer value) indicates that the corresponding parameter value should be preserved. RESOVL sends these values to the Command Executor which calls the actual reset routine.

5.3.1.3.19 Overlay CZOV19 - Module INIDAT - Command INITIALIZE

INIDAT is called upon an INITIALIZE command and, in addition, during the system preparation sequence. It displays the current values of six system parameters (seed and crucible diameter, boric oxide weight, and the densities of the solid crystal, the semiconductor melt, and the boric oxide melt),

and permits the operator to either accept them by pressing "Return" only, or to enter new data. Negative values, which are invalid in any case, are trapped, and the parameters are converted back to their internal storage format. In order to facilitate diameter evaluation, the system uses the squares of the diameter values, and densities in grams per cubic millimeter. Finally, INIDTA checks the minimum height of the boric oxide encapsulant melt (i.e., the height of a cylinder of molten boric oxide with the specified mass which fills the entire cross section of the crucible), and sets the boric oxide weight to zero if it is too small to be handled properly by the Diameter Evaluation routine (compare chapter 5.3.2.2.3).

5.3.1.3.20 Overlay CZOV20 - Module PLOTOV - Command PLOT

The module PLOTOV permits to link Variables or memory locations specified by absolute addresses to one of the eight Plot Channels. An approach similar to the one used in DEBUG0 is applied to separate Variable and address inputs; Variables must be in INTEGER*2 format in order to be displayed, whereas this format is implicitly assumed for memory locations specified by their absolute addresses. PLOTOV scans the input string for name/address and channel information, and requests data if applicable. Further processing of the PLOT command is done by the Command Executor to which a pertinent message is dispatched.

5.3.1.3.21 Overlay CZOV21 - Module CLEARO - Command CLEAR

Two versions of the CLEAR command are supported by CLEARO, namely, the Unconditional, and the Selective Clear. CLEARO scans the input line for the name of a Variable, and assumes that an Unconditional Clear is issued if no Variable name is found. An operator reconfirmation is requested, and a prompt for a Variable name is issued if the operator indicates he did not want an Unconditional Clear. The Variable name is processed as usual (with FINDAD), and a command message is sent to the Command Executor when the operator acknowledges his entries.

5.3.1.4 The Command Executor - Task CMMDEX

The Command Executor receives command messages from two sources, namely, from the Command Interpreter, and from the Command File Input Task (compare Fig. 17). The special FORTRAN-iRMX-80 Interface Routines used (compare chapters 5.2.1.1 and 5.2.1.2) automatically advance these messages to the Command File Output Task which eventually records them in the Command Output file. The Command Executor's commission is to process each command message, and to execute several other procedures which have do be run in regular intervals.

CMMDEX runs once every second; its timing is indirectly derived from the Timer Task FXTIME. The first action of CMMDEX after it performed a few initialization subroutine calls (but not the first action after it starts running every second) is to receive a command message if there is one. In most cases, there will be none; the approach of using command messages has, however, the advantage that these messages will be queued by the operating system in the order in which they were issued if more of them are generated than can be processed. Therefore, it is possible to have CMMDEX process only one command message every second without losing commands; in the worst case, the command execution may be delayed by a few seconds.

5.3.1.4.1 Command Message Processing

CMMDEX first decodes a command message if one was received. The first byte of each command message holds a command code which consists of a major Mode (corresponding to "SET" or "DEBUG") in the high four bits of the byte, and a Switch (e.g., for "SL" or "Continuously") in the low four bits. These values are separated, and control is vectored to the proper processing sequences.

Mode = 1 and 2 - SET and CHANGE Internal parameter

Mode values of 1 and 2 correspond to SET and CHANGE commands, respectively, which apply to Internal parameters. CMMDEX first determines the address where the specified setpoint is to be stored: There are two arrays for the setpoints of the Internal parameters, STPNT0 and STPNT1, which correspond to the left and right setpoint columns displayed on the console screen. STPNT0 always holds the setpoint values which are actually used by the various controller routines, and which are the ones which are normally ramped by CMMDEX. There is, however, an important exception to this rule if an Internal

parameter is controlled, e.g., the heater temperatures in diameter controlled operation modes. In this case, CMMDEX stores the output of its ramping generator in STPNT1 rather then in STPNT0; the LOGICAL Function CNTRL (which is actually an assembly language subroutine) returns ".TRUE." in this case. CMMDEX sets a memory location according to the (setpoint) variable type (all Internal parameters are two-byte integers) and enters a sequence of code which is also used for SET and CHANGE commands applying to Variables, and by the Conditional Command Executor algorithms. A flag (L) is used to branch to the Conditional Command Executor code after the common sequence; although this approach is certainly not consistent with structural programming techniques, it is the most efficient way with regard to the large number of local memory locations whose contents are used inside and outside the common code.

Having issued a warning if a SET or CHANGE command was entered while the CGCS is in monitoring mode, i.e., not controlling the puller, CMMDEX determines the current contents of the target locations, and converts them to floating-point format if necessary. For CHANGE commands (Modes 2 or 10), a new final value is calculated by adding the message input value to the current target location contents; for SET commands, it is directly derived from the value passed with the command message. The magnitude of the resulting value is checked if it has to be stored to integer locations which have a limited numeric range; the result is set to the permitted maximum with the correct sign, and an error message is issued, if an overflow is detected. Similarly, diameter, temperature, and power limit setpoints are checked for negative values; the above result is set to zero and an error message ensues if any of these setpoint values is found to be negative.

The processing of all SET and CHANGE commands continues in a Command Executor code sequence called Ramping Preparation: The CGCS is able to ramp up to twenty independent parameters; all ramping control structures are therefore arrays with twenty elements each. Each channel holds an address, a variable type, and starting, final, increment, and breakpoint values; the latter four in floating-point notation to guarantee the necessary resolution and dynamic range. CMMDEX assigns a ramping channel according to availability to a parameter or Variable which is to be ramped; only the Variable address indicates which datum is handled in which channel. In order to prevent confusions if a SET or CHANGE command was issued for data which are currently being ramped, CMMDEX checks first whether the address passed is already used by one of the ramping channels, in which case this particular channel is updated. Otherwise, CMMDEX searches for an unused ramping

channel (unless all channels are used or a transition time of zero was entered, in which cases ramping is bypassed, and the final value is stored immediately at the target address). The status of a ramping channel is determined by RMPFLG which is zero if a channel is not used. It is set to the number of an Internal parameter (1 to 9), or to -1 if a Variable is ramped. The Ramping Preparation code stores the current and final values of the parameter to be ramped; an increment value is calculated by dividing the difference between the initial and the final values by the ramping time in seconds, and a breakpoint, by multiplying the absolute value of the increment by 1.1 and adding a small number. Finally, CMMDEX stores the final values of Internal parameter setpoints in the corresponding array (unless the parameter is being controlled), and continues with the Ramping Executor sequence.

Mode = 3 - Macro command, Unconditional CLEAR

A Mode value of 3 indicates either a Macro command (Switch = 0), or an Unconditional CLEAR command (Switch = 1).

The Macro name passed with a Macro command message is expanded into a full Macro file name (i.e., ".CMD" is appended). A possibly active Macro command is preempted (with a QUITCM call), and a pertinent message is issued. CMMDEX tries to open the Macro file; the QUITCM call is repeated if the old Macro file was not yet closed by the Command File Input Task or if the file could not be opened due to a temporary shortage of pool memory (which may happen under adverse conditions). An explicit error message ("Macro ... doesn't exist") is generated if the new file was not found, and the internal disk error message is output in case of any other error. CMMDEX reads the first 16 bytes of the new Macro command file and checks whether the first two bytes hold zeros, and the next two, the version code of the currently used system. The Macro command is cancelled if the first condition is not met, and a message referring to a "restricted command set" is issued if the system versions do not match. The flag DEBUGE is set in this case; it indicates to the Command File Input Task that all commands which refer to absolute memory locations, i.e., all commands with a command Mode value greater than 7, must be discarded (compare chapter 5.3.1.6). In any case, IOFLAG(4) is set to indicate to the Command File Input task that there is an active Macro file, and the flag RUNTIM activates the alarm clock timer. An internal counter, MACPRO, is set to a starting value of 4. This value will be decremented by 1 during each of the subsequent passes of CMMDEX, once every second, until it finally reaches zero; the checking of Conditional commands is inhibited while MACPRO is not zero (compare

chapter 5.3.1.4.5). Finally, CMMDEX writes the name of the Macro command into the top line of the output screen, and generates a message which indicates that the execution of this Macro was started.

The Unconditional CLEAR command is processed very straightforwardly: The eight Conditional Command flags and the Conditional Command counter are reset, and a pertinent message is issued.

Mode = 4 - MODE

CMMDEX outputs a "New mode:" message upon receipt of a MODE command, and triggers a Data Dump in the Documentation Output. Subsequently, it checks for the following mode changes which require special initialization:

(1) Change from "Monitoring" to any controlled mode: In order to avoid transients when the CGCS takes over from the analog system, all measured values of the Internal parameters have to be duplicated to the corresponding setpoint locations in STPNT0. Simultaneously, all ramping flags and the ramping counter are reset. It is therefore not possible to transfer a constant or ramped Internal parameter setpoint from uncontrolled to controlled mode.

(2) Change from not diameter controlled modes ("Monitoring", "Manual") to any diameter controlled mode: CMMDEX checks in this case whether the Diameter setpoint is currently being ramped, stops its ramping if it is, and copies the current actual Diameter value to both Diameter setpoint locations.

(3) Changes between modes in which certain Internal parameters are controlled: All twenty ramping channels are scanned to find a ramped Internal parameter which was not controlled in the previous mode but is controlled in the new mode, or vice versa. In the first case, the output of the Ramping Generator is directed to the second setpoint array STPNT1 rather than to the first one, STPNT0, and the current value of the affected element in STPNT0 is copied to STPNT1 in order to avoid transients; in the second case, the ramping is stopped.

CMMDEX finally sets the Mode flag used by the remainder of the system, and jumps to a sequence in the Ramping Executor which outputs the Mode information in the "fixed" part of the console screen.

Mode = 7 - RESET

A RESET command is processed by a simple call to the subroutine RESET which is part of the assembly language module SHAPE (compare chapter 5.3.2.2).

Mode = 9 and 10 - SET and CHANGE Variable

Essentially, the algorithms described for Mode = 1 and 2 are used in treating Variables.

Mode = 11 - IF and Selective CLEAR

A SWITCH value of 0 represents a Conditional (IF) command, a value of 1, a Selective CLEAR.

CMMDEX can handle up to eight Conditional Macro commands; Conditional commands which are issued while already eight commands are pending are ignored, and a pertinent error message is issued. After a free storage location was found for the new command, the two relation code values and the Variable type information are extracted from the command message byte in which they were stored, and the comparison value, the Variable address, and the name of the Macro command which is to be executed conditionally are written to the proper locations for use by the Conditional Command Executor.

In case of a Selective CLEAR command, CMMDEX compares the Variable addresses stored for all currently active Conditional Command Channels to the Variable address passed with the command message. A channel is deactivated (by resetting its status flag), and a "Conditional Macro cleared" message is issued if matching values are detected. (This may happen more than once if several Conditional commands referred to the same Variable.)

Mode = 14 - PLOT

CMMDEX stores the address passed with the command message in the element of an address array which is determined by the Plot Channel number specified with the command.

Mode = 15 - DEBUG

The DEBUG command uses a Switch value which can have the values 2 through 5. Upon a DEBUG Continuously command (Switch = 2), the variable address and type are stored in locations of the pertinent Debug arrays whose index is determined by the Debug Channel number (1 to 4). The DEBUG Modify command (Switch = 3) is processed by storing the correct number of bytes at the specified address. The DEBUG Resume and Suspend commands, finally, are executed by calls to the proper interface routines.

5.3.1.4.2 The Ramping Executor

This part of the Command Executor is accessed after the treatment of any command, and if no command was received at all. The first commission of the Ramping Executor has nothing to do with ramping yet: CMMDEX tests the flag RESTDO(2), and writes the operation mode to the "fixed" part of the console screen whenever RESTDO(2) is found set, resetting the flag simultaneously. (This flag is set within the routine FRAME to enforce data output to the console screen after it was cleared; compare chapter 5.3.1.3.8.)

CMMDEX waits subsequently for a Flag Interrupt which is triggered once every second when a flag is set by the Analog Controller Task ANACNT which, in turn, is triggered directly by the Timer Task FXTIME. (The structure of the FORTRAN-iRMX-80 Interface software prohibits that several tasks be triggered in parallel by the Timer output.) The subsequent parameter ramping is therefore done in relatively regular intervals of one second, no matter how long the processing of input data took.

For each active ramping channel (ramping flag not equal to zero), CMMDEX tests whether the absolute value of the difference between the current setpoint and the final value is already less than the breakpoint value which was determined during the ramping preparation. The current setpoint is set to the final value in this case, and ramping of this channel is disabled. Otherwise, the increment is added to the current setpoint, and the current setpoint is stored in memory in the proper format in either case.

5.3.1.4.3 Floating-Point Conversion of Measured Data

All measured analog data are primarily stored and processed as two-byte integers. Unfortunately, these values are hardly suitable for use in Conditional Macro commands because they have to be scaled to be meaningful. This is done by the Command Executor during each pass.

5.3.1.4.4 DEBUG Data Retrieval

During each pass, CMMDEX reads four bytes at the addresses specified with each active DEBUG Continuously command, and stores them in an array from which they will eventually be output by the Measured Data Output Task.

5.3.1.4.5 Conditional Command Executor

The Conditional Command Executor sequence within CMMDEX is executed only if the internal counter MACPRO holds zero. MACPRO is preset whenever a new (Conditional or unconditional) Macro command has been started (compare chapter 5.3.1.4.1), and is reset to zero within several seconds. (The timing of CMMDEX is slightly corrupted when a Macro command is being activated, due to the relatively time-consuming disk accesses involved in this process. It is therefore not possible to specify the exact duration of the delay enforced with MACPRO.) Disabling Conditional command checking temporarily while a new Macro is being started guarantees that at least the first command of the new Macro can be executed (provided its relative time is zero or 1 seconds) without being preempted by a possibly concurrently activated Conditional command. (The first command in each file which must not be preempted prematurely should therefore be a CLEAR command at a relative time of 0 or 1.)

For each of the eight potentially pending Conditional Macro commands, the value stored at the specified Variable address is read (using the algorithm of the SET/CHANGE commands, compare chapter 5.3.1.4.1) and compared to the constant passed with the command. If the result of the comparison matches the specified relation(s), the Macro command is invoked using the code sequence of the Macro command processing described in chapter 5.3.1.4.1. The message "Conditional Macro started" is output before control is transferred to the standard Macro command processing. This results in a possibly confusing

sequence of messages if the Conditional Macro preempts an active Macro command:

"Conditional Macro started" - output by the Conditional Command Executor.

"Command Macro preempted" - refers to the old Macro.

"Executing Macro ... " - gives the name of the new Conditional Macro started.

5.3.1.4.6 Data Dump to the Documentation File

Data dumps are generated by the subroutine DUMPDT which is invoked by the Command Executor's main routine CMMDEX. DUMPDT continues execution if either the Dump Flag DUMPFL is set, indicating a Data Dump request, or if the number of one-minute Flag Interrupts has been encountered which was specified as a Dump interval when the Documentation file was opened. DUMPDT generates three output lines which contain 21, essentially measured, system parameters identified by two-character mnemonics (compare chapter 4.6). A pointer array is used to assign data from the (floating-point) array REALDT to the proper output locations, and output is written to a buffer which is pre-loaded with the identification text frame when the CGCS is read into memory. The three buffers each of which outputs seven parameter values are written in turn to line 24 of the console screen (which is, in fact, not usable for permanent display since it is cleared during each console input request), using the standard Console/Documentation file output routines STROUT (compare chapter 5.2.2.9). In this case, it is not the output on the CRT screen we are interested in, but its duplicate, tagged with the time information, in the Documentation output. (Line 24 of the CRT screen is cleared by a concluding line of spaces to keep the console screen tidy.) Finally, DUMPFL is reset in any case in order to be ready to accept a new Dump request.

The FORTRAN module which holds DUMPDT contains, in addition, a small routine DUMP which can be called by any task which wants to trigger a Data Dump. In addition to setting DUMPFL to ".TRUE.", this routine also sets the flag byte TIMINT which is normally set by the Timer Task FXTIME after the interval specified when the Data file was opened, triggering the output of a data record to the Data file. Therefore, an additional record is entered into the Data file when DUMP is called. (The status of TIMINT does not matter if no Data file is open and active.)

5.3.1.4.7 Analog Output to a Chart Recorder

The subroutine PLOTPR which is called next by the Command Executor Task CMMDEX prepares data for analog output. PLOTPR does not output these data, though; the latter commission is done by the Analog Data Controller ANACNT.

First, PLOTPR calculates the "expanded" temperature, growth rate, and diameter and crucible position errors which were specially provided for chart recorder output. This procedure involves, in general, proper scaling, limiting to maximum and minimum values, and adding of an offset if required. Next, PLOTPR retrieves the contents of the eight locations pointed to by the Plot Channel address array elements. It calculates the absolute values of these data, and provides a message on the console (and in the Documentation output) if a Plot Channel changed its sign since the last pass of PLOTPR. The resulting eight INTEGER*2 values are stored with interrupts disabled in order to prevent problems caused by the real-time environment; the (assembly language) subroutines DISINT and ENINT disable and enable interrupts, respectively.

5.3.1.4.8 Program Code Integrity Check

In order to improve the chances to detect inadvertent modifications of the CGCS program code - due to hardware failures or to software problems -, the routine MEMCHK is called at the end of the CMMDEX code, before the task resumes its infinite loop. MEMCHK calculates a signature byte for each 256 byte page within the main CGCS program code, and compares it to the signature obtained during the previous pass. An error message is issued if the two signatures are found to be different, i.e., if one of the 256 bytes checked changed its contents, and the signature byte kept in memory is updated to the new value. The output of error messages is suppressed during the first pass of MEMCHK, immediately after the CGCS was loaded, to permit the array of signature bytes to be initialized properly. With each call of MEMCHK, five (5) 256-byte pages are processed; MEMCHK loops to the beginning of its surveillance area after it arrived at its end. The about 150 memory pages which are monitored by MEMCHK are therefore tested once every 30 seconds. The memory check comprises the entire CGCS resident code area; for obvious reasons, it could neither include the data locations nor the overlay area. Still, the memory check encompasses about 70 percent of the entire memory area, and it is easily possible that MEMCHK might detect a damaged program code byte before it has been executed (with conceivably disastrous results).

5.3.1.5 The Measured Data Output Task - Task MEASDO

The task MEASDO provides all periodically updated output to the console. It is not synchronized to any other system task but loops continuously through its code. In order to prevent MEASDO from monopolizing the system (which would have the effect that tasks with a lower priority could never be executed), there are deliberate waits built into the task: At eight roughly equidistant locations within MEASDO, the task calls the subroutine WTOUTP which, in turn, executes a wait operation for a specifiable number of iRMX-80 time units. (One iRMX-80 time unit is 50 ms.) The number of time units for which WTOUTP waits is kept in the system Variable INTRVL which can be modified with SET, CHANGE, or DEBUG Modify commands. The smaller this number is, the faster runs MEASDO obviously; the minimum value of INTRVL is 1. (An INTRVL value of 0 halts MEASDO indefinitely and irreversibly.) The current default INTRVL value is 10; this value gives satisfactory response during normal system operations. It is, however, recommendable to reduce the INTRVL value during adjustments of the analog data acquisition hardware. The INTRVL value should be restored to its standard value when full growth control is required, in order to protect the system from being overloaded.

The infinite task loop of MEASDO is entered after two initialization calls to FORTRAN-iRMX-80 Interface routines; it starts with a check of the flag RESTDO(3) which "short-circuits" the task if it is set, thus inhibiting data output. Next, MEASDO copies all data which are to be output into internal memory locations. This is necessary because most of the data locations are protected by software interlocks to prevent them from simultaneous accesses by several tasks. MEASDO would unduly block the data locations and, in consequence, all tasks which also attempt to access them if it monopolized them during the lengthy output operations; on the other hand, repeated access and release operations would impose an unacceptable overhead. The approach for copying the data locations one by one, rather than using a program loop, may bewilder experienced FORTRAN programmers. For the given number of items, however, the technique chosen is faster and more code efficient than a loop.

The actual output operations follow one standard approach: The current (integer) data value to be output is compared to the value of the same item during the previous pass of MEASDO. The integer value is multiplied by its scaling factor and written to the console only if the two values are different, thus preventing the repeated output of constant data. In order to enforce the data output regardless of whether an item

was updated or not, a loop is provided which sets all items of the "old" array to the contents of the corresponding items in the "current" array, plus 1, if the flag RESTDO(1) is set. Evidently, the two arrays will hold different values for each output item after this procedure.

After having output all items, MEASDO sets the "old" output data array to the contents of the "current" data, and tests whether output is required for DEBUG Continuously commands. The scrolled output area is limited to four lines (18 through 21), and line 17 is cleared if Debug output was activated since the last pass; the scrolled area is set to five lines (17 through 21) if Debug output was deactivated. MEASDO writes the address and the memory contents for each active Debug channel into line 17; prior to the actual data output, the screen area corresponding to the respective channel is cleared (overwritten with spaces) if the Debug output mode was changed. This procedure clears channels which are deactivated, and it removes the previous output completely if the interpretation of Debug data and therefore the number of displayed digits was changed.

5.3.1.6 The Command File Input Task - Task CMFINP

The Command File Input Task CMFINP reads commands from a Macro file and sends them with the proper timing to the Command Executor.

Each command in a Macro file is tagged with the time at which the particular command is to be executed, relative to the time of the call to the Macro command. This is accomplished by a combined action of the Command File Input and the Timer Tasks:

The Command Executor sets the flag RUNTIM after each call to an existing Macro command file, which indicates to the Timer Task that the seconds counter #2 is to be started (compare chapter 5.2.4.1). The contents of this counter are compared by the Timer Task to an "alarm clock" value once every second, and a flag is set if the counter value is equal to or greater than the "alarm clock" setpoint. The "alarm clock" value is set by the Command File Input task according to the relative time of the next command which is to be processed; CMFINP is, in turn, triggered by the "alarm clock" flag.

The task loop of CMFINP starts with a wait for an "alarm clock" flag interrupt. Since CMFINP sets the initial "alarm" value to zero, such an interrupt will happen immediately when the timer is enabled by CMMDEX upon a Macro command. CMFINP

reads one record from the Macro command file. The task branches if a disk error occurred, if the end of the file was encountered, or if a command which refers to absolute memory locations was read from a Macro file generated under a different system version. In the first two cases, an "End of Macro command file" message is output, the input from the file is disabled (IOFLAG(4) is reset), the Macro file is closed, the Macro name in the first line of the console screen is deleted, and the timer #2 is disabled by resetting RUNTIM (which is called INPACT in CMFINP). CMFINP reports "Macro command not executable" and reads the next command if the third exception condition was detected. For all valid commands, CMFINP sets the "alarm clock" to the execution time of the next command, and loops back to the initial wait. The command message is dispatched to the Command Executor immediately after the "alarm clock" interrupt.

This handling of Macro command files accounts for possibly confusing sequences of system messages: CMFINP reads the next command immediately after having sent the preceding one to the Command Executor. CMFINP messages may therefore appear on the screen even before the last command was processed by the Command Executor.

5.3.1.7 The Command File Output Task - Task CMFOUT

The Command File Output Task CMFOUT receives command messages from the Command Executor. It appends the relative time of the command, i.e., the difference between the current value of the (first) seconds counter in the Timer Task FXTIME (compare chapter 5.2.4.1) and the value which this counter had when the START command was issued. This time information is stored in the first two bytes of a 16 bytes record, followed by the command message proper. CMFOUT traps Macro commands which are not recorded on purpose (compare chapter 4.5), writes the record to the Command Output file, and disables itself if the command was an END command.

5.3.1.8 The Disk Output Task - Task DSKOUT

The task DSKOUT which resides in the FORTRAN module DSKDAT is in charge of output to the Data file. DSKOUT collects all data in a buffer which are to be recorded, and writes this buffer to the disk.

5.3.2 The Process Controller

5.3.2.1 The PID Controller Routine FRPIDC

The actual process control approach used in the digital CGCS is, to a large degree, based on approaches used with conventional analog techniques. In particular, the system uses PID (Proportional-Integral-Derivative) controllers for the closed-loop control of various parameters. In contrast to an analog system, however, where separate controller hardware is required for every control loop, the CGCS contains only one generic PID controller routine which performs (with different parameters) the following functions (compare chapter 4.1.2):

(1) Motor Speed Control: The outputs which determine the speeds of the four puller motors (seed and crucible lift and rotation) are controlled according to the differences between the corresponding setpoints and the actual speed values. This approach permits to compensate for motor controller imperfections such as nonlinearities or offsets.

(2) Temperature Control: The power output by the heater(s) is controlled to maintain the heater temperature setpoint(s).

(3) Diameter Control: The heater temperature setpoints are adjusted to provide a minimum diameter error.

(4) Crucible Position Control: The lift speed of the crucible is controlled to reduce to zero the difference between a calculated crucible position setpoint and the pertinent actual value.

The PID Controller routine is, indeed, part of the FORTRAN-iRMX-80 Interface programs, and is linked with the CGCS code from the library FXUTIL.LIB. It is discussed here, though, because of its great impact on the operation of the controller routines proper.

FRPIDC is based upon high-speed integer algorithms. Its output is calculated in several steps: Within the first step, the error E is derived from the setpoint S and the actual value A by:

$$E = S - A \qquad (1)$$

Subsequently, an intermediate result X is calculated according to the following algorithm:

$$X = E \cdot P/256 + (IE \cdot I)/IS + DE \cdot D/256 \qquad (2),$$

with P, I, and D, the proportional, integral, and derivative multipliers, respectively. IE represents the error integral, i.e., the sum of all error values encountered since the PID controller went into operation; IS is a scaling factor which can be either equal to 256 or to 65536 (2^8 and 2^{16}, respectively), depending on the value of a control flag. DE, finally, is the difference to the preceding error:

$$IE \cdot I = E_0 \cdot I + E_1 \cdot I + \ldots + E_n \cdot I \qquad (3)$$

$$DE = E_n - E_{n-1} \qquad (4)$$

The three terms in (2) are scaled by 256 (or by 65536) in order to permit effective proportional, integral, and derivative multipliers with an absolute magnitude of less than one. (The above values of the scaling factors were chosen because a multiplication or division by a power of two imposes the least time and code overhead.)

The integral component is calculated by accumulating the sum of the error values, each multiplied by the integral multiplier, in a four byte (32 bit) memory location. (This approach is less sensitive to abrupt changes of the integral multiplier I which may happen during the tuning of the system, compared to accumulating the error sum and multiplying it by the integral multiplier.) Depending on the magnitude of the integral scaling divisor IS, either the least significant byte, byte 0 (for IS = 256), or the two least significant bytes, bytes 0 and 1 (for IS = 65536), of this internal sum are discarded when the integral component of X is determined. The integral component of X is the value of the next two bytes, 2 and 1, or 3 and 2, respectively, which is rounded according to the most significant bit of the discarded byte(s), and set to the maximum positive or negative value if IS = 256 was chosen and the accumulation of the integral extended into the fourth byte (byte 3). Optionally, the resulting two byte integral component may be compared to a limit value; the entire four byte integral is modified to return an integral component which is exactly equal to the limit value (with the sign of the four byte error integral) if the integral component would otherwise exceed the limit.

The intermediate result X is calculated by first adding the proportional and the derivative components, and finally, the integral component. X may be limited to any arbitrary range if the user chooses so; for a given limit L, X results in

$$\begin{aligned} X &= -(L + 1) && \text{if } X < -(L + 1) \\ X &= L && \text{if } X > L \\ X &= X && \text{otherwise} \end{aligned} \qquad (5)$$

This limiting operation is independent from the limiting of the integral component although the same limit parameter is used. A default value for L is assumed in either case (with L = 32767) if either no limit was specified, or if a negative limit value was given.

In order to improve the dynamic response of the PID routine, a "wind-up protection" feature was included. This feature prevents the error integral IE from overflowing when a limit condition is incurred. Without "wind-up protection", the error integral would continue accumulating in this case, which might become particularly disturbing if a scaling factor IS of 256 is used and the integral extends into the highest byte. The error integral would subsequently require a very long time to recover from the previous condition even if the error already changed its sign, which might obviously lead to control instabilities. The "wind-up protection" can be explicitly activated by the user; it becomes only effective when a limit condition is incurred. In this case, the internally stored four byte error integral can be adjusted in either one of two ways: it can either be set to a value resulting in a (two byte) integral component equal to the difference between the limit value and the sum of the proportional and the derivative components (mode A), or it may be adjusted to return an integral component equal to the positive or negative limit value, as demanded by the error integral's sign (mode B). Thus, the PID controller is forced to remain in its active operation area; the intermediate result X reacts immediately or almost immediately to a decrease of the error rather than after the delay otherwise inherent with the reduction of the error integral. Activating the "wind-up protection" overrides the integral component limiting function. The dynamic behavior of the PID controller under its various operation modes is discussed in more detail in Appendix 14.

Finally, the intermediate result is submitted to two additional adjustments: First, it is multiplied by a factor which is a (positive or negative) power of two, and second, a bias value B is added:

$$M = B + 2^G \cdot X \qquad (6)$$

The result, M, is output at last. The scaling factor 2^G was provided to permit an adjustment of the controller's output to various devices. Some devices require, e.g., less than 16 bit signed data, in which case a negative G value can be used to dispose of the least significant G bits. It could also be used for restricting the output signal to a certain range. A positive G could increase the overall gain of the controller; with regard to accuracy, this is, however, not recommended.

The bias input, finally, centers the output of the PID controller around the bias value B.

This handling of the bias value permits to introduce a nonlinear PID controller response by means of two stacked PID controllers. The setpoint and actual data inputs of both controllers are connected in parallel; the output of the first is used as a bias input for the second. Single controller operation can be achieved by setting all multipliers of one of the two stacked controllers to zero. (Which one does not matter since they are exchangeable.) Nonlinear control is possible if different parameters are attributed to both controllers, and the output limit L is set to a value considerably less than 32767 for one of them. For an error resulting in an intermediate signal X of the output limited controller less than ±L, the output of the two controllers is the sum of the outputs of either controller, and the resulting P, I, and D values are the arithmetic sums of the corresponding parameters of both controllers. In the output limited mode, the limited controller contributes only its limit value whereas the other controller continues operating in its linear range; the P, I, and D parameters of the two-controller system are thus equal to the parameters of the controller remaining active. It seems, however, advisable to introduce some kind of wind-up protection at least for the output-limited controller in order to permit a fast response of the system to sudden error changes. Similarly, two stacked controllers can be operated with different limit values and properly chosen P, I, and D multipliers in order to provide, for example, integral control with a wider limit margin than the faster proportional and derivative components, which may contribute to an improved reliability of the controller under noisy conditions.

The CGCS features the stacked PID controller approach for its three diameter control loops, and for crucible position control. Particularly for diameter control, the second approach mentioned above - different limits for the slow integral and for the fast proportional and derivative controller components - proved advantageous because it can very effectively prohibit large and fast excursions of the heater temperature setpoints due to the inevitable noise superimposed on the calculated diameter value.

Essentially, the operation of the digital PID controller routine is akin to any analog PID controller. The time constants in the integral and derivative parts of the controller function are determined by the frequency 1/T with which the controller runs. In the linear region of the controller (no limit incurred), eqs. (2) through (6) may be re-written as:

$$M = B + 2^G \cdot P'[E + I'/(P'\cdot T) \int_0^T Edt + (D'\cdot T)/P'\cdot dE/dt \,] \quad (7)$$

with P', I', and D', the proportional, integral, and derivative multipliers of eq. (2) times their appropriate scaling factors.

The parameters for the PID controller are kept in a 12 byte array. The first two bytes of this array must be accessed from FORTRAN as INTEGER*1 locations (one byte integers), the remainder, as INTEGER*2. The following data are kept in this array:

Byte 0: Gain Multiplier Exponent G

Byte 1: Control Byte, containing switches for:

- Integral Component Scaling: 0 ... IS = 256; 1 ... IS = 65536
- Output Limit: 0 ... No Explicit Output Limit; 1 ... Output Limit = +/- L
- Wind-Up Protection: 0 ... Off; 1 ... On
- Wind-Up Protection Mode: 0 ... Integral set to L-(P+D); 1 ... Integral set to ±L
- Integral Component Limiting: 0 ... Off; 1 ... On

Byte 2+3: Bias Value B

Byte 4+5: Proportional Multiplier P

Byte 6+7: Integral Multiplier I

Byte 8+9: Derivative Multiplier D

Byte 10+11: Limit Value L

The control byte permits to set the operation mode of the PID routine, namely, the scaling of the integral component, the output limiting operations, and the wind-up protection. The decimal values of the control byte listed below correspond therefore to the following operations:

CNTL	IS	Limit	Wind-Up Prot.		Integr.Lim.
0	256	±32767	OFF		±32767
1	65536	±32767	OFF		±32767
2	256	±L	OFF		±32767
3	65536	±L	OFF		±32767
4	256	±32767	ON	Mode A	±32767
5	65536	±32767	ON	Mode A	±32767
6	256	±L	ON	Mode A	±32767
7	65536	±L	ON	Mode A	±32767
8	256	±32767	OFF		±32767
9	65536	±32767	OFF		±32767
10	256	±L	OFF		±32767
11	65536	±L	OFF		±32767
12	256	±32767	ON	Mode B	±32767
13	65536	±32767	ON	Mode B	±32767
14	256	±L	ON	Mode B	±32767
15	65536	±L	ON	Mode B	±32767
16	256	±32767	OFF		±L
17	65536	±32767	OFF		±L
18	256	±L	OFF		±L
19	65536	±L	OFF		±L
20	256	±32767	ON	Mode A	*
21	65536	±32767	ON	Mode A	*
22	256	±L	ON	Mode A	*
23	65536	±L	ON	Mode A	*
24	256	±32767	OFF		±L
25	65536	±32767	OFF		±L
26	256	±L	OFF		±L
27	65536	±L	OFF		±L
28	256	±32767	ON	Mode B	*
29	65536	±32767	ON	Mode B	*
30	256	±L	ON	Mode B	*
31	65536	±L	ON	Mode B	*

* Wind-up protection overrides integral limiting.

Wind-up Protection mode A entails that the integral component is set to the limit value minus the sum of the proportional and the derivative components if the output exceeds the limit; in mode B, the integral component is set to the positive or negative limit value, as appropriate. Compare Appendix 14 for a detailed discussion of the operation modes of FRPIDC.

ROUTINE FRPIDC:

Routine Type: Assembly language subroutine; reentrant.

Initialization: none

Routine Call:

CALL FRPIDC (data, parameter)

with: data: Name of a 6*INTEGER*2 array holding:
1. The actual measured value (I).
2. The setpoint (I).
3. The controller output value (O).
4. The previous error (*).
5. and
6. The error integral (*).
Data marked "*" are set by FRPIDC and should not be changed.

param.: Name of a 6*INTEGER*2 array holding:
1.(low) The gain multiplier exponent (I).
1.(high) A flag byte (I).
2. The bias value (I).
3. The proportional multiplier (I).
4. The integral multiplier (I).
5. The derivative multiplier (I).
6. The limit value (I).

Required Stack: 22 bytes.

5.3.2.2 The Diameter Controller - Task DIACNT

5.3.2.2.1 The Diameter Controller Routine Proper - Module DIACNT

The FORTRAN module DIACNT constitutes the main routine of the Diameter Controller Task. This task is triggered every ten seconds by a "flag interrupt" generated by the Timer Task FXTIME (compare chapter 5.2.4.1).

DIACNT uses a command message of its own in order to perform automatic RESET and MODE commands. The commands issued by DIACNT are not recorded in the Control Output file, and the command message issued by DIACNT is returned to this task after having been processed by the Command Executor. This implies that DIACNT has to retrieve this message from its response exchange before it is permitted to use it again. The

FXACPT call at the beginning of the infinite loop in DIACNT serves exactly this purpose.

A sequence immediately following this subroutine call checks for changes into diameter controlled mode while the Diameter Evaluation routines have not been reset yet. The following steps ensue if such a condition is detected:

(1) DIACNT issues a RESET command which sets the weight and crystal length grown locations to zero, and generates a pertinent message.

(2) It stores the current MODE value (which can be 2, 3, or 4) in an auxiliary location INTMOD, sets the operation mode to Manual, and marks this condition with a SHSTOL value of -3. The remainder of the task loop is skipped.

    This procedure triggers a Reset operation when the Command Executor runs the next time. The actual diameter value is still meaningless because the Diameter Evaluation routine runs only <u>after</u> a Reset; it will be ignored because the operation mode is still set to Manual.

(3) During its next pass, ten seconds later, DIACNT will set the MODE value back to the value saved in INTMOD; it will duly execute the Diameter Evaluation routines which will return a meaningful diameter value now, but it will skip the Diameter Control sequence.

    The Command Executor runs only after DIACNT has finished (because its priority is much lower), and it will find a meaningful Actual Diameter value which it can copy to the Diameter Setpoint locations (because of the change to diameter controlled mode).

(4) Only at the third pass, twenty seconds after a change to diameter controlled mode without preceding Reset was detected, DIACNT will resume its standard operation.

Under regular operating conditions, DIACNT copies the operation mode value into a local location in order to avoid confusions if the mode is changed while this task is running.

Subsequently, DIACNT retrieves from the array of analog input data the Crucible Position and the Differential Weight values, and converts the latter into floating-point notation, scaling it with the proper scaling factor. This datum is first submitted to the subroutine ANOMAL (compare chapter 5.3.2.2.2) which performs an anomaly compensation if required, and subsequently, to the function SHAPE (compare chapter 5.3.2.2.3).

SHAPE calculates a Diameter value (returned in DIAMET), and, in addition, the Length grown (scaled with the same factor as the Seed Position input data), and a Crucible Position setpoint (in SCRUCP) which is in the same format as the Crucible Position input data. (Several other auxiliary values are returned by SHAPE which are primarily intended for testing and debugging purposes.) SHAPE provides a status value in SHSTAT which is evaluated after its execution; corresponding (error) messages are issued the first time a new SHSTAT value is returned, and the operation mode is set to Manual if either a Zero Seed Lift Speed or a Speed Overflow error was detected. With the exception of changes to or from a Zero Seed Lift Speed condition and of an Oxide Height Overflow, a data dump to the Documentation output and an additional record in the Data file are triggered. (The data dump is omitted on purpose in the two cases mentioned because either error may be reported repeatedly although the state of the process did not change significantly; a data dump at each of these occasions would have been a waste of paper and/or disk space.) DIACNT tries to re-activate SHAPE after a Speed Overflow error with a call to the subroutine REACTV which is part of the assembly language module holding SHAPE; after six succeeding unsuccessful attempts, DIACNT decides that the problem is too serious to deal with it on its own, and disables SHAPE permanently (until a RESET command is issued again).

Finally, DIACNT enters the actual diameter controller sequence: While in Monitoring or in Manual mode, DIACNT has nothing to control. The routine resets, however, the Error Integral locations of the PID Data arrays and the Previous Error values (compare chapter 5.3.2.1) to zero. This is done to provide a defined environment when diameter control is activated.

In any one of the diameter controlled modes (Diameter, Diameter/ASC, and Automatic, compare chapters 4.1.1 and 4.3.3), DIACNT has to generate three Heater Temperature setpoints. Each of these setpoints is obtained from two stacked PID controllers which permit to obtain a non-linear control response (compare chapter 5.3.2.1). The first PID controller receives the proper current Heater Temperature setpoint (i.e., the value obtained from operator or Macro commands) from the STPNT1 array as a Bias value; its output is used as a Bias for the second PID controller. All six diameter controllers use the actual Diameter and the Diameter setpoint as inputs. The output of the second PID controller of each Heater channel is stored in the setpoint array STPNT0.

An additional control loop is executed in Automatic mode: The crucible lift speed is controlled according to the difference

between the actual Crucible Position and the pertinent setpoint calculated by SHAPE. Two stacked controllers are used for this commission, too; similar to the diameter controllers, the Crucible Lift Speed setpoint input by the operator (or from a Macro file) is used as a Bias value for the first controller, whose output is fed to the Bias input of the second controller. The second controller's output is stored as an actual Crucible Lift setpoint.

The approach chosen for the PID controllers in DIACNT, namely, passing the "manual" setpoint through the controller routines via the Bias inputs, has various advantages: Since the "manual" setpoint can be chosen to lie close to the actually required controller output, the PID controllers need only make small modifications to the "manual" setpoint, which improves the accuracy and the dynamic response of these routines. Furthermore, it is possible to limit the output of the controllers to lie within a relatively small range around the Bias value. This prevents, for example, the Diameter controller from totally turning off the heater if the actual crystal diameter seems to be much smaller than the pertinent setpoint, which may easily happen particularly during cone growth. In fact, a smooth transition from manual to diameter controlled growth may be obtained if the controllers' PID parameters are ramped from zero to their final values, or if the Limit values are initially set to zero and slowly ramped to their intended final values. There is, indeed, hardly any limit set to the control schemes which may be obtained from dynamically modifying the parameters of the controllers provided.

5.3.2.2.2 Anomaly Compensation - Routine ANOMLY

Prior to the evaluation of the diameter, the Differential Weight value derived from the A/D converter can optionally be submitted to a compensation for anomaly effects. According to the conventional anomaly compensation approach, a corrected Differential Weight X can be calculated from the "raw" weight Y by solving the differential equation

$$X = (Y - b \cdot X')' \qquad (8)$$

where X' is the first derivative of X with respect to time. Equation (8) can be re-written as

$$b \cdot X'' + X = Y' \qquad (9).$$

In the CGCS, we expanded the above approach to:

$$b \cdot X'' + a \cdot X' + X = Y' \qquad (10)$$

Numerically, the above differentiations have to be replaced by differences. With X_0, the current Differential Weight, X_1, the previous value, and X_2, the previous but one, we can write:

$$X_0' = X_0 - X_1$$

$$X_1' = X_1 - X_2 \qquad (11)$$

$$X_0'' = X_0' - X_1' = X_0 - 2 \cdot X_1 + X_2$$

Substituting eqs. (11) into eq. (10) results in a linear equation for X_0 which can be solved as:

$$X_0 = \frac{Y' + (a + 2 \cdot b) \cdot X_1 - b \cdot X_2}{1 + a + b} \qquad (12)$$

The "raw" Differential Weight Y' is input directly from the analog differentiator circuitry. A corrected Differential Weight X_0 can be calculated from the "raw" value Y and the previous results X_1 and X_2 according to (12). The FORTRAN subroutine ANOMAL evaluates X_0 from eq. (12) if the Mode value is greater than 2 (i.e., in Diameter/ASC and Automatic modes; compare chapters 4.1.1 and 4.3.3); otherwise, it sets X_0 equal to Y'. In addition, ANOMAL stores its X_1 value in X_2, and the X_0 value thus determined, in X_1, in order to have proper previous results for the next pass.

5.3.2.2.3 Diameter Evaluation Algorithms - Routine SHAPE

The assembly language routine SHAPE constitutes the heart of the diameter evaluation algorithms. SHAPE calculates the following data:

(1) A crystal Diameter value which is derived from the Differential Weight which previously may have been submitted to anomaly compensation. SHAPE takes into account the buoyancy in the boric oxide melt.

(2) The current height of the boric oxide melt in the crucible.

(3) The Crystal Length grown.

(4) A Crucible Position setpoint which is used for determining the crucible lift speed in Automatic mode.

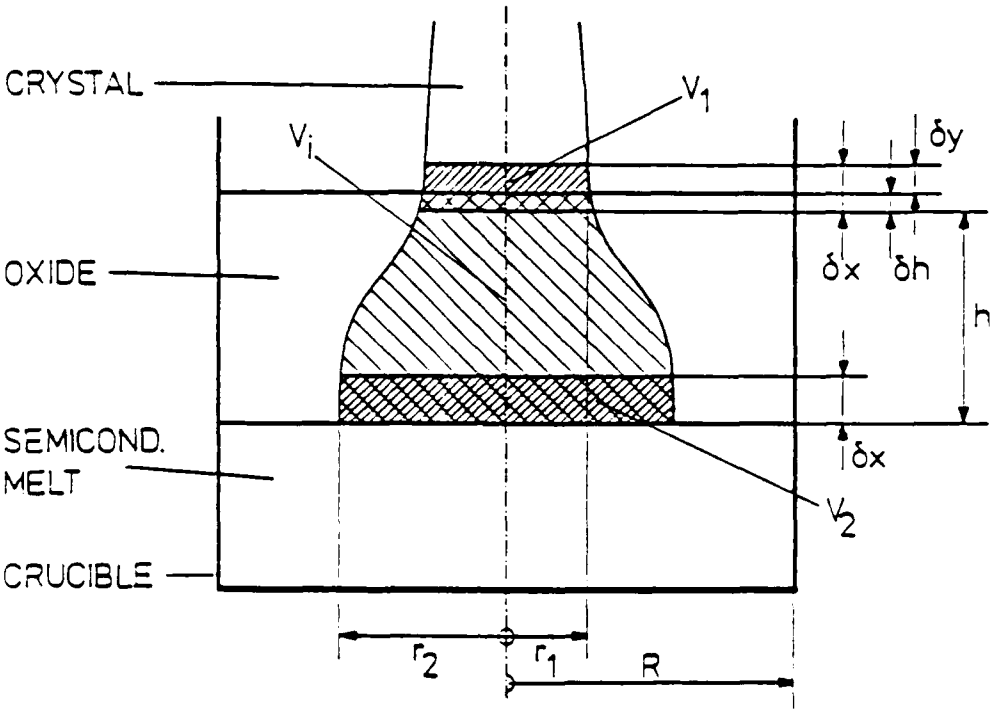


Fig. 18: Growth of a crystal partially immersed in an oxide encapsulant melt.

When the length of a crystal grown increases by δx, a portion of the crystal whose length is δy emerges from the boric oxide melt. The two differential lengths are not necessarily equal since the height h of the boric oxide melt may have been changed by δh due to a change of the crystal volume immersed (compare Fig. 18). We can write:

$$\delta y = \delta x - \delta h \qquad (13)$$

The height h of the oxide melt can be determined from the oxide melt volume V_m and the volume V_i of the immersed part of the crystal, with R, the radius of the crucible:

$$V_m + V_i = R^2 \cdot \pi \cdot h \qquad (14)$$

During the major part of a crystal growth run, V_m is constant. Towards the end of the run, however, the semiconductor melt starts retracting from the crucible wall, resulting in a disk of molten gallium arsenide in the center of the crucible. The height of this disk remains roughly constant but its diameter decreases. The gap between this disk and the crucible is

filled by boric oxide, causing the effective oxide volume (i.e., the volume measured from the extension of the top surface of the semiconductor melt disk upwards) to decrease. Differentiation of eq. (14) results in:

$$\delta V_m + \delta V_i = R^2 \cdot \pi \cdot \delta h \qquad (15),$$

with

$$\delta V_m = - \epsilon \cdot \delta V_2 \cdot (d_x / d_{xm}) \qquad (16)$$

and

$$\delta V_i = \delta V_2 - \delta V_1 \qquad (17),$$

where

$$\delta V_2 = r_2^2 \cdot \pi \cdot \delta x \qquad (18)$$

and

$$\delta V_1 = r_1^2 \cdot \pi \cdot \delta y \qquad (19).$$

The parameter ϵ in eq. (16) is equal to zero during regular growth, and equal to 1 if the melt contraction described above has reached its full extent, i.e., if the surface of the semiconductor melt does not drop any more. The effective boric oxide volume is reduced in this case by the volume of semiconductor melt required to grow the differential cylinder δV_2; d_{xm} and d_x stand for the densities of molten and solid semiconductor material, and r_1 and r_2 are the radii of the crystal at the oxide surface and the melt-crystal interface, respectively. With d_o, the density of the boric oxide melt, the change of the crystal's weight δW can be written as:

$$\delta W = \delta V_2 \cdot (d_x - d_o) - \delta V_1 \cdot (d_x - d_o) + \delta V_1 \cdot d_x =$$

$$= \delta V_2 \cdot (d_x - d_o) + \delta V_1 \cdot d_o \qquad (20),$$

The differential cylinder close to the semiconductor melt contributes to the weight only with the difference of the crystal and oxide densities, due to buoyancy; the differential cylinder which emerged from the oxide melt had previously a weight proportional to $(d_x - d_o)$ which is now proportional to d_x only.

With eqs. (13) and (15) to (19), we can express δy as a function of δx:

$$\delta y = \frac{R^2 - \beta \cdot r_2^2}{R^2 - r_1^2} \cdot \delta x \qquad (21),$$

with

$$\beta = 1 - \epsilon \cdot (d_x / d_{xm}) \qquad (22).$$

Note that β is equal to 1 during regular growth, and it approaches 0 when the melt recession starts (because the ratio of densities is close to 1). With eq. (21), we can re-write eq. (20):

$$\frac{\delta W}{\pi \cdot \delta x} = r_2^2 \cdot (d_x - d_o - \beta \cdot d_a) + R^2 \cdot d_a \qquad (23),$$

with an "adjusted oxide density" d_a

$$d_a = d_o \cdot \frac{r_1^2}{R^2 - r_1^2} = d_o \cdot \frac{1}{(R^2/r_1^2) - 1} \qquad (24).$$

Eq. (23) permits to calculate the square of r_2:

$$r_2^2 = \frac{\frac{\delta W}{\pi \cdot \delta x} - R^2 \cdot d_a}{d_x - d_o - \beta \cdot d_a} \qquad (25)$$

With the Differential Weight ($\delta W/\delta t$) and the Growth Rate

$$v = \delta x / \delta t \qquad (26),$$

we finally can write for eq. (25):

$$r_2^2 = \frac{\frac{(\delta W/\delta t)}{\pi \cdot v} - R^2 \cdot d_a}{d_x - d_o - \beta \cdot d_a} \qquad (27)$$

The Growth Rate v is determined by the combined effects of the Seed Lift Speed v_s and the speed v_d with which the gallium arsenide melt drops:

$$v = v_s + v_d \qquad (28).$$

For a length δx of crystal grown, the semiconductor melt in the crucible will drop by δz in order to provide the crystal mass solidified while the crystal is within the regular growth regime. The melt level will hardly drop any more when the melt contraction towards the end of the growth run started. Since the total mass must be constant, we can write:

$$R^2 \cdot \pi \cdot d_{xm} \cdot \delta z = r_2^2 \cdot \pi \cdot d_x \cdot (\delta x + \delta z - v_c \cdot \delta t) \qquad (29),$$

with d_{xm} and d_x, the densities of the semiconductor melt and the solid crystal, respectively, and v_c, the actual Crucible Lift Speed. We can solve eq. (29) for δz and can, finally, obtain:

$$v = \frac{v_s - v_c}{1 - \alpha \cdot \frac{r_2^2 \cdot d_x}{R^2 \cdot d_{xm}}} \qquad (30).$$

The constant α is equal to 1 if (30) is obtained as an exact solution of eq. (29). In a heuristic way, however, assigning values different from 1 to α can help to compensate for non-ideal effects caused by the crucible shape and/or surface tension. Values of α greater than 1 can compensate for a decrease of the crucible diameter close to its bottom; in contrast, α can be set to values less than 1 to take into account the receding of the gallium arsenide melt during the final stages of the growth process, an effect which obviously more than compensates for the beveling of the crucible. The surface of the melt does, in effect, hardly drop any more when the semiconductor melt starts receding from the crucible walls because the material used up by the growing crystal is mainly supplied by reducing the diameter of the melt disk rather than its height. This corresponds to a crucible with infinite diameter, or to an α value of 0. At the end of the body growth, α may simply be ramped down to 0, starting at the point when melt recession usually begins. (A Variable named ALPHA is provided for this purpose. It is initialized with the value 1 but may be modified with the standard SET or CHANGE commands.)

The constant ϵ defined in eq. (16) follows exactly the opposite behavior, compared to α: it is equal to zero during the regular growth, and assumes a value of 1 at the end of the process. It appeared therefore to be a reasonable approach to set

$$\epsilon = 1 - \alpha \qquad (31)$$

within the SHAPE software.

Substituting v from eq. (30) into eq. (27), and solving the resulting equation for r_2^2, results in

$$r_2^2 = \frac{\frac{dW/dt}{\pi \cdot (v_s - v_c)} - R^2 d_a}{d_x - d_o - \beta \cdot d_a + \frac{dW/dt}{\pi \cdot (v_s - v_c)} \cdot \frac{\alpha}{R^2} \cdot \frac{d_x}{d_{xm}}} \quad (32)$$

Obviously, the currently grown crystal diameter can be determined as twice the square root of the left side of eq. (32). The result obtained from solving eq. (32) is submitted to two correction steps which confine the effects of possible numeric errors due to non-ideal input signals: Negative values of the result are trapped and converted to zero (because the square of a real magnitude like the crystal diameter cannot become negative), and values which exceed the maximum permitted crystal diameter (100 millimeters) are limited to the maximum. The latter is accomplished with a small FORTRAN module DIALIM, which is actually part of the main Diameter Controller task, DIACNT, and which is called from the assembly language routine SHAPE. SHAPE returns an INTEGER*2 diameter value in the Variable IDIAMT; this value is converted to floating-point notation, scaled properly, and stored in the Variable DIAMET by the Command Executor.

The evaluation of eq. (32) implies a division by the difference of the seed and crucible lift speeds, v_s and v_c, respectively. Obviously, this value must not be equal to zero to permit a valid calculation of the diameter. SHAPE checks therefore this difference right at the beginning of its operations; it skips the remainder of its code and sets an error flag if it detects a zero value. The error flag is monitored by DIACNT; suitable action is taken, and an appropriate message is output in the case of an error (compare chapter 5.3.2.2.1).

In order to solve eq. (32), the square of the radius of the crystal at the surface of the oxide melt, r_1^2, has to be known. This entails that the actual height of the boric oxide melt, h, and the total volume of the crystal immersed in the boric oxide, V_i, are also known; the latter parameters are required for calculating the optimum crucible position. SHAPE determines these data by keeping a table of crystal diameter squares in an array DIATAB. (In fact, SHAPE operates with diameter rather than radius squares; with the exception of a factor of 4 in the denominator of the first term in the numerator of eq. (32) and in the corresponding term in the denominator of eq. (32), the algorithms within SHAPE are identical to those above. We used radii rather than diameters in the above derivation in order to avoid naming confusions with the densities.)

Since SHAPE can only store the shape (i.e., the diameter) of the crystal at discrete length positions, an interpolation approach had to be developed which permits an approximate evaluation of the crystal's diameter at any arbitrary position. An obvious method would have been a linear interpolation between the stored diameter values. For the application in mind (where the squares of diameter values are more often required than the plain diameters), a linear interpolation of the squares of the diameter data proved to be considerably more efficient. Assuming that the square of the diameter or the radius is a linear function of the position x within the crystal, we can write

$$r^2 = k \cdot (x + x_o) \qquad (33),$$

with k and x_o, constants determined by the crystal's shape. (We return again to radii rather than diameters in order to match the above nomenclature.)

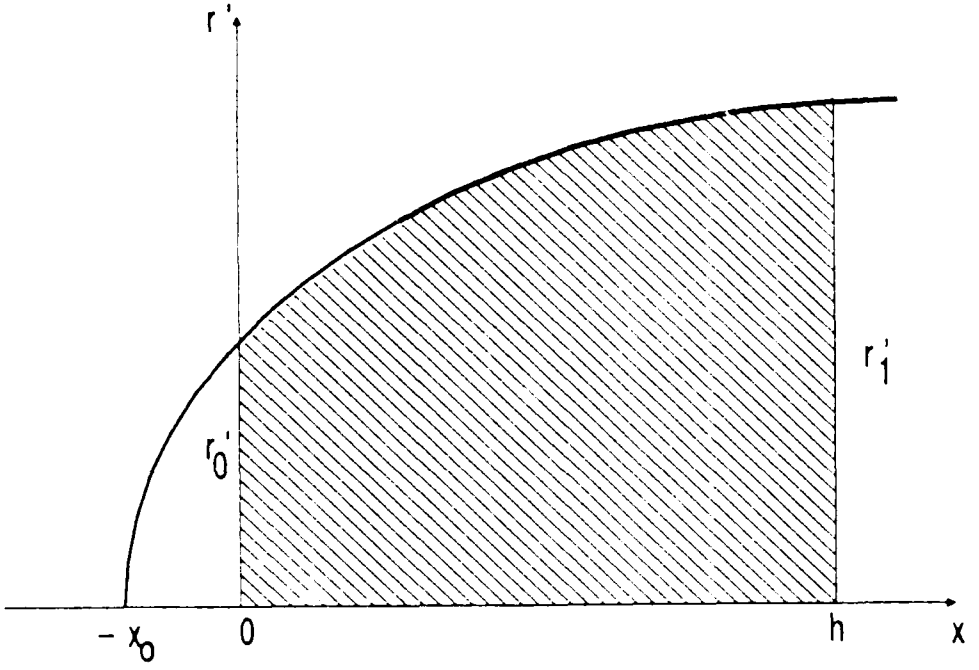


Fig. 19: Volume of a paraboloid section.

Equation (33) means that a section of the crystal is approximated by a section of a paraboloid. Figure 19 depicts the radius r' of a section with the height h_s; the radius at the

bottom of the section is r_0', and on top of the section, r_1'. Applying eq. (33) to $x = 0$ and $x = h_s$, respectively, permits to replace the constants k and x_o with r_0', r_1', and h_s:

$$k = \frac{r_1'^2 - r_0'^2}{h} \tag{34}$$

$$x_o = h_s \cdot \frac{r_0'^2}{r_1'^2 - r_0'^2} \tag{35}.$$

The volume V of the paraboloid section obtained from rotation of the shaded area in Fig. 19 around the x axis can be calculated as:

$$V = \pi \cdot \int_0^{h_s} r^2 \cdot dx \tag{36}$$

With (33) through (36), we obtain finally:

$$V = \frac{\pi \cdot h_s}{2} \cdot (r_0'^2 + r_1'^2) \tag{37}$$

During regular crystal growth, SHAPE accumulates the volume within one "slice" of the crystal by adding the volumes of "differential" cylinders with the diameter of the crystal calculated in the previous pass and with a height determined by the difference of the crystal length values for the current and the previous pass. This volume increment may be negative during meltback conditions, or if the new length value was less than the previous one due to noise superimposed on the seed and crucible position signals. A "slice" boundary is detected when the two byte integer representation of the crystal length exceeds a multiple of 64, which corresponds to a length difference of approximately 1.17 mm. A new "slice" is added to SHAPE's image of the crystal in memory which is kept in DIATAB, a 64 element floating-point array of squares of diameters, calculating the square of the diameter of a cylinder with a height equal to the distance from the previous slice boundary, and a volume equal to the sum of "differential" volumes accumulated since then. (Due to noise and the limited resolution of the crystal length, the height of this cylinder may be slightly greater than 64 length counts.) (An earlier approach to approximate the crystal by slices of paraboloids proved to be unstable because errors of the previously calculated diameter squares propagated into the newly calcul-

ated data when eq. (37) was solved.) All entries in DIATAB are shifted up one step, and the new diameter square is stored as the crystal's bottom diameter. The top entry is lost.

In order to prevent erratic diameter square average values from being entered in the table and from eventually corrupting the diameter calculation when the slice in question arrives at the encapsulant surface, the FORTRAN subroutine CHKDTB was provided which is part of the module DIACNT but called from SHAPE. CHKDTB compares the absolute value of the difference between the preceding and the current squares, and adjusts the new value to differ by not more than the specified limit (which is kept in the Variable XTLSHP) from its predecessor. This approach allows for greater absolute and relative diameter fluctuations in stages where the crystal diameter is small (e.g., in the early cone sections) and where such fluctuations are quite normal; it is more restrictive within the full-diameter crystal body. The data stored in DIATAB are in square millimeters; XTLSHP must therefore be set to the maximum permitted difference between the squares of the diameters (in millimeters) of two adjoining crystal sections.

During meltback conditions, i.e., when the crystal length decreases rather than increases with time, the entries in the array of diameter squares are shifted down one step when a slice boundary is reached; the top entry is reduplicated.

A subroutine of SHAPE, CALCSD, uses the entries in DIATAB to calculate the square of the diameter of the crystal at the surface of the boric oxide melt (corresponding to r_1^2 in our calculations). Since the position of the oxide melt surface relative to the crystal depends on the total height of the oxide melt, which is, in turn, a function of the total crystal volume immersed in the boric oxide, the melt height and the immersed volume must be re-calculated in each pass of CALCSD. The following procedure is used in CALCSD (compare Fig. 20):

The portion of the crystal immersed in the boric oxide melt is divided into slices of uniform height h_s whose radii (or rather, diameter squares) are stored in DIATAB. The top and bottom slices are obviously exceptions to this rule. The bottom slice is the portion of the crystal grown since the last slice boundary was encountered; the height of the top slice is determined by the position of the oxide surface.

In order to determine r_1^2, CALCSD assumes that the encapsulant melt height did not change since the last pass. CALCSD first checks whether the boric oxide height is less than 75 millimeters, i.e., less than the length of the portion of the crystal whose diameter squares are stored in DIATAB. The oxide

height is limited to the maximum permitted value, and an error output is triggered if this is not the case. In order to prevent the "Oxide Height Overflow" error from either being reported every ten seconds, or from eventually hiding any other error condition which is flagged by the same parameter of SHAPE, SHSTAT, a counter byte is incremented for every occurrence of this error; SHSTAT, in contrast, is set to indicate the problem only when the counter wraps around to zero after 256 increments. Depending on the number of iterations required in CALCSD, this may happen every 5 to 20 minutes if the condition persists continuously.

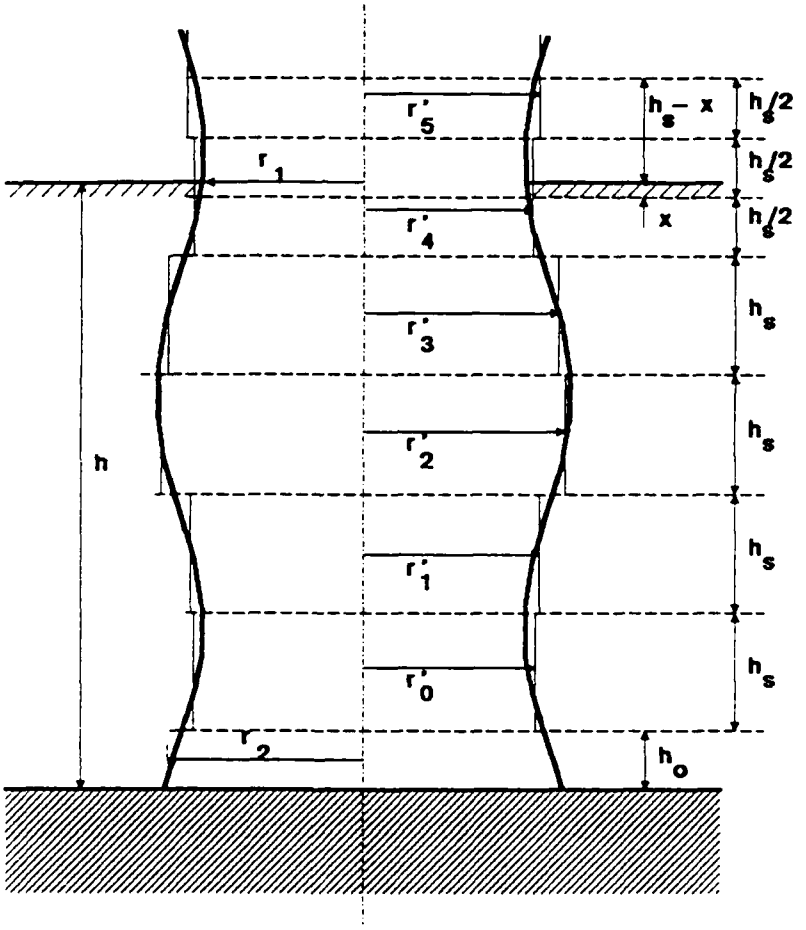


Fig. 20: Interpolation algorithm for the evaluation of the crystal diameter at the boric oxide encapsulant surface, and of the volume immersed.

Having checked the oxide height, CALCSD subtracts the height of the bottom slice, h_o, from the previous melt height h and determines by a simple modulo operation the distance x the melt surface lies above the center of the last slice which is immersed to more than 50 percent of its height. The square of r_1 is obtained from a linear interpolation of the two adjacent entries in the table (in Fig. 20, from the squares of r'_4 and r'_5), i.e., by an interpolation which assumes a paraboloid shape of the current section. Using the centers of the slices rather than the slice boundaries as top and bottom surfaces of a paraboloid section guarantees a better accuracy.

The melt height h, however, may have changed since the previous pass if the volume added at the growth zone was not equal to the volume withdrawn from the boric oxide. In order to determine the current value of h from eq. (14), the immersed crystal volume V_i must be known. For a crystal with n slices covered by the oxide melt to at least 50 percent of their height, we can calculate V_i according to:

$$V_i = V' + \pi \cdot h_s \cdot [r'_0{}^2 + r'_1{}^2 + \ldots + r'_{n-1}{}^2 + r'_n{}^2/2] + \pi \cdot x \cdot (r'_n{}^2 + r_1{}^2)/2 \qquad (38).$$

(V' is the volume of the currently grown section of the crystal with the height h_o.)

Equation (14) permits now to calculate a new melt height value (assuming the oxide volume V_m and the crucible radius R are known) which is compared to the previous height. CALCSD returns if both values differed for less than one height unit in INTEGER representation (approximately 0.02 mm); otherwise, the procedure is repeated from the calculation of r_1 on. CALCSD is left, though, if a certain number of iterations (currently, 5) was not sufficient, which constitutes a protection against "freezing" in the case of bad convergence.

During the major part of the growth run, the oxide volume V_m in eq. (14) is, indeed, known and constant. Towards the end of the run, however, the active boric oxide volume starts decreasing as the semiconductor melt recedes from the crucible wall, and the resulting gap is filled with boric oxide. Therefore, V_m has to be corrected after each pass in this regime for the apparent oxide volume loss δV_m according to eq. (16):

$$V_m = V_{mo} + \Sigma\, \delta V_m = V_{mo} - \Sigma\, (\epsilon \cdot \delta V_2 . d_x/d_{xm}) \qquad (39),$$

where V_{mo} is the initial boric oxide melt volume.

One more task of SHAPE is the evaluation of a Crucible Position setpoint which also enters into the calculation of the Length Grown. The apparent weight of the growing crystal, W, can be written as

$$W = W_o + W_x - (V_i - V_{io}) \cdot d_o \qquad (40),$$

where W_o is the measured initial weight at the beginning of the growth run, and W_x, the actual weight of the crystal grown. The last term in eq. (40) takes into account the buoyancy in the boric oxide melt. The mass W_x has been withdrawn from the contents of the crucible, essentially by lowering the surface of the semiconductor melt. Towards the end of the growth run, however, the semiconductor melt volume required to grow the crystal is supplied by shrinking the diameter rather than the height of the semiconductor melt. The volume thus obtained is identical to the boric oxide volume lost according to eqs. (16) and (39). The crucible must be raised by a distance z in order to keep the semiconductor melt surface at the same location within the puller:

$$W_x = R^2 \cdot \pi \cdot z \cdot d_{xm} - \Sigma\ \delta V_m \cdot d_{xm} \qquad (41)$$

(Note that δV_m is negative; the contribution of the right term in eq. (41) is therefore either zero - during regular growth - or positive.)

The density of the melt, d_{xm}, is different from the density of the crystal, d_x. The crucible position setpoint, z_s, can be calculated from eqs. (40) and (41) with the initial crucible position z_o:

$$z_s = z_o + z =$$

$$= z_o + \frac{W + V_i \cdot d_o - (W_o + V_{io} \cdot d_o) + \Sigma\ \delta V_m \cdot d_{xm}}{R^2 \cdot \pi \cdot d_{xm}} \qquad (42)$$

We may substitute with eqs. (14) and (39) for the immersed volumes V_i and V_{io}, and we obtain with the melt height h_o at the beginning of the growth run:

$$z_s = \frac{W}{R^2 \cdot \pi \cdot d_{xm}} + h \cdot \frac{d_o}{d_{xm}} + \frac{\Sigma\ \delta V_m}{R^2 \cdot \pi} \cdot \left(1 - \frac{d_o}{d_{xm}}\right) +$$

$$+ \left(z_o - h_o \cdot \frac{d_o}{d_{xm}} - \frac{W_o}{R^2 \cdot \pi \cdot d_{xm}}\right) \qquad (43)$$

The term in the second line of eq. (43) is an initialization constant. The calculated Crucible Position setpoint is returned by SHAPE in the INTEGER*2 Variable SCRUCP.

The actual position z_a of the crucible as obtained from the Crucible Position potentiometer may be different from z_s; with the initial and actual seed position values x_o and x_a, respectively, the length l of the crystal can be calculated as:

$$l = (x_a - x_o) + (z_s - z_a) \qquad (44).$$

An accordingly determined Crystal Length value is returned by SHAPE in the Variable ILENGT (in INTEGER*2 notation); it is eventually converted to floating-point format and scaled by the Command Executor and stored as LENGTH.

Note that different approaches are used for the calculation of the actual Growth Rate and of the Crucible Position setpoint and Length Grown values. In the first case, the Growth rate is derived from (measured) speed values, while the Crucible Position setpoint calculation is based on the weight of the crystal. Although this approach may appear redundant, it was indispensable in order to obtain an acceptable accuracy of the results. (In general, it is preferable to use an input value which constitutes already an integral magnitude (such as the crystal weight), rather than calculating an integral; similar considerations apply to derivative data such as speeds.)

A Meltback condition is indicated by SHAPE and subsequently reported by DIACNT if the Crystal Length value calculated was decreased by more than one "slice", i.e., by more than 1 mm. This may be due to any effect which reduces the distance between the seed and the semiconductor melt surface, whether it was caused by a movement of the seed, or of the crucible. A "Regular growth resumed" message is similarly issued after a RESET command, upon the detection of a non-zero Seed Lift speed after a "Zero seed lift speed" error, or if the Crystal Length was increased again by more than one "slice" after a Meltback condition.

SHAPE is called as an INTEGER*1 Function from FORTRAN; the Differential Weight is passed to it as a parameter. It returns an integer flag which indicates the status of its operation. The following values are currently defined:

3 ... Oxide Height overflow.
2 ... Seed Lift speed is zero - no diameter calculated.
1 ... Meltback.
0 ... Regular growth.
-1 ... Speed overflow - RESET or REACTV call required.
-2 ... SHAPE is not yet initialized - RESET required.

All other parameters are passed to and from SHAPE via memory locations in COMMON blocks most of which can be accessed as Variables.

5.3.2.2.4 The Initialization of the Routine SHAPE - Routine RESET

Although the subroutine RESET is logically part of the Command Executor Task CMMDEX, it is kept in one assembly language module together with SHAPE, and it is discussed here. The essential commission of RESET is to prepare SHAPE for its operations. RESET has to be called before usable Diameter, Length grown, and Crucible Position setpoint values can be obtained from SHAPE if the status value returned by SHAPE is negative (compare chapter 5.3.2.2.3). This value is negative

(a) After the start of the system, before RESET was called the first time, and

(b) After a "Speed overflow" error which happens if SHAPE is no more able to update its stack of crystal "slices". This is the case if less than one diameter value per "slice" is available, corresponding to speeds in excess of 400 mm/h.

(In the latter case, a call to the subroutine REACTV may be sufficient to re-establish proper operation of SHAPE; compare chapter 5.3.2.2.5.)

The following items are initialized by RESET:

(1) The initial values of the crystal Weight, the Seed, and the Crucible Positions.

(2) The Diameter Square Table DIATAB is filled assuming a cylindrical crystal with a diameter equal to the Seed Diameter which was specified with the INITIALIZE function. The initial immersed crystal volume is calculated accordingly.

(3) The initial melt height is derived from the Melt Weight and Density values obtained from INITIALIZE, and from the above initial immersed volume.

(4) The parameter ALPHA (compare chapter 5.3.2.2.3) is set to 1, and the encapsulant volume "lost" by melt recession (compare eq. (16)) is reset to zero.

5.3.2.2.5 The Re-Activation of SHAPE - Routine REACTV

The subroutine REACTV is kept in one module together with SHAPE. It permits to safely resume the operation of SHAPE after a Speed Overflow error which was not likely to have caused significant changes to the volume of the crystal immersed in the boric oxide melt. REACTV simply resets the internal location which is used to accumulate the volume of the crystal "slice" grown since the last pass of SHAPE, and it activates SHAPE again by resetting its status byte.

5.3.2.3 The Analog Data Controller - Task ANACNT

5.3.2.3.1 The Analog Controller Routine Proper - Module ANACNT

The main part of the Analog Data Controller task is kept in the FORTRAN module ANACNT. The following operations are done by the Analog Data Controller task:

(1) ANACNT requests from the A/D Converter board the A/D converted values for 25 analog input channels, and it preprocesses the Weight value by subtracting the offset weight determined by RESET.

(2) It controls the power output for three heaters, running PID controller routines with the Heater Temperature setpoints and actual values as inputs.

(3) It generates similarly the control output to the four motors.

(4) It provides input from and output to the Motor Direction relay circuitry, and it takes care of the Controller Selection output.

(5) It writes the control data determined above and the Plot data collected by the Command Executor (compare chapter 5.3.1.4.7) to the D/A Converter hardware.

ANACNT runs once every second, being triggered directly by the Timer task FXTIME (compare chapter 5.2.4.1). Immediately after having been re-activated, it sets an auxiliary seconds flag, SECFLG, which, in turn, sets to work the Command Executor.

The first major operation of ANACNT is obtaining input data from the A/D Converter board. This is done within the assembly language subroutine ANAINP (compare chapter 5.3.2.3.2). The ANAINP call is skipped if the flag TEST is set to -1 (compare chapter 4.7.2). ANAINP returns the input from the Converter hardware in the 25 element array ANALOG, as two-byte integer data. (The contents of this array may be patched with arbitrary simulation data which can even be ramped if required if ANAINP was disabled with TEST.)

Having read the input data, ANACNT prepares the data and parameter locations for the seven PID controllers which are run by this task. The three Temperature controllers (one for each zone of a three-zone heater) use the pertinent Heater Temperature setpoints and actual values as input data; the Power Limit setpoint serves as a common Limit value for all three controllers. The controllers' Bias values are kept at zero.

A totally different approach is used for the Motor Speed controllers: On principle, the motor controller hardware could be driven directly by the D/A converted speed setpoints. Due to nonlinearities and offset errors within the motor controller hardware, this approach would result in unsightly differences between the speed setpoints and the actual speeds. In order to alleviate this problem, PID controllers were provided for each of the four motor channels which determine the actual output signals which are eventually fed to the analog motor drivers after their D/A conversion. Due to the potentially conflicting requirements imposed upon these controllers, a special approach was taken: For operation modes where the absolute accuracy of the motor speeds is less demanding but a fast reaction to speed setpoint changes is required, a feedforward can be provided via the Bias input which is set to the pertinent speed setpoint in this case. A correction of the offset between the speed setpoints and the actual motor speeds may be introduced by appropriate programming of the PID controller's parameters. While this mode can be advantageously used during the heating and dipping stages, it may offer insufficient stability during the growth process proper where hardly any motor speed changes are required but where speeds

should be kept as stable as possible. The feed-forward function which is liable for possible control oscillations can be disabled in this regime either totally or partly by setting a factor θ (i.e., the proper element of the array THETA) to a sufficiently small value. THETA is introduced into the four motor speed controllers according to

$$M = \theta \cdot B/256 + 2^G \cdot X \qquad (45),$$

where M is the final output of the PID controller, B, its Bias value (in this particular case, the proper speed setpoint), G, the Gain Multiplier Exponent, and X, the output of the PID routine proper (compare chapter 5.3.2.1, eq. (6)). Setting THETA equal to 256 provides therefore full feed-forward, while a THETA value of 0 results in plain PID operation. The controller is inherently slower in this mode but can more easily be tuned for a stable operation. (The scaling by 256 was chosen because of the same reasons as for the PID parameters, namely, in order to provide a multiplication by a factor less than 1 with INTEGER arithmetics. The INTEGER multiplication is done with the subroutine IMULT which is called by ANACNT.)

ANACNT prepares the inputs to the PID routines in any case; it runs the PID controllers only if the CGCS is in charge of the puller. In contrast, it resets the Previous Error and Error Integral locations of all controllers while the system is in Monitoring mode, and it provides the motor speed setpoints for output by the D/A Converter board. (This provision was made in order to permit an easier test of the output hardware. The motor speed control signals are thus available at the outputs of the digital controller in any case.)

A special treatment is required for the motor speeds: Due to the offset usually introduced by the PID controllers, a non-zero output signal results even if the corresponding setpoint was actually set to zero. Although this non-zero output signal might only compensate for an opposite offset of the hardware, it would prevent the Motor Direction Relay controller routine MOTDIR (compare chapter 5.3.2.3.3) from switching the motors actually off. ANACNT branches therefore according to the motor speed setpoint values, and provides a zero output explicitly when required.

Having trapped possibly negative Temperature Controller output values (there is no negative heater power), ANACNT copies the internal array of analog input data, ANALOG, to the array ANADAT, shifting the contents of the ANALOG array by one element. This was done to guarantee that the important input data (i.e., the first 17 elements of ANADAT) are actually sampled at the same time. The first element in ANALOG was

measured at the end of the previous call to the Analog Data Input routine ANAINP (compare chapter 5.3.2.3.2), and it is therefore approximately one second older.

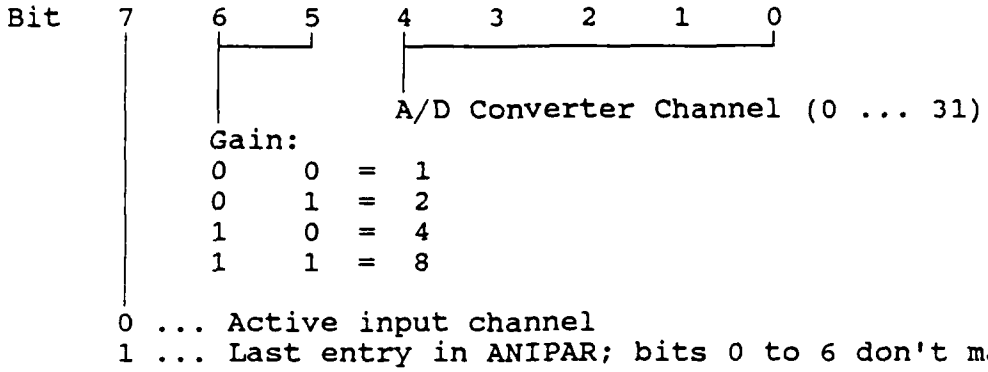
Finally, ANACNT calls the Motor Direction Relay controller routine MOTDIR (compare chapter 5.3.2.3.3), and writes the fifteen Analog Output values in the array ANAOUT (three temperatures, four motors, and eight chart recorder output channels) to the D/A Converter board, calling the Analog Data Output routine ANAOPT (compare chapter 5.3.2.3.4). Both operations are skipped if TEST is set to -1.

5.3.2.3.2 The Analog Data Input Routine ANAINP

ANAINP is an assembly language routine which reads data from the A/D Converter board in a random access mode, and submits the values obtained to digital low-pass filtering.

In order to permit random input of data from the hardware, ANAINP uses a special parameter array ANIPAR which consists of two bytes for each channel. It is, therefore, very easy to connect a logical data channel within the CGCS to an arbitrary hardware channel and to modify the gain and filtering parameters of any channel by changing the contents of ANIPAR. In addition, the number of input channels actually read is not built into ANAINP but derived from the parameter array ANIPAR: The operation of ANAINP is terminated, and the routine returns to the calling task, when the most significant bit of an odd element of ANIPAR is set, corresponding to any negative value. (The remainder of the parameter byte does not matter.) It is therefore essential that at least one negative value is provided in ANIPAR lest ANAINP might indefinitely continue reading data; since the output is stored in an array which is specified as the second parameter of ANAINP, this data input would exceed the boundaries of the array and eventually overwrite important data. The two parameter bytes per channel in ANIPAR hold the following information:

ANIPAR (2n + 1): (n = 0, 1, 2, 3 ...)



ANIPAR (2n + 2): (n = 0, 1, 2, 3 ...)

Low-pass filter flag, determines the cut-off frequency of the digital low-pass filter routine:

Value	Cut-Off Frequency (Hz)
0	infinite
1	0.1150
2	0.0461
3	0.0213
4	0.0103

ANAINP supposes that a valid result is held by the A/D Converter hardware for the first input channel. Correspondingly, the last action of ANAINP prior to its return, and one of the actions of the initialization routine for ANAINP, ANAINI, is to prepare and trigger the conversion of the first input channel. Since approximately one second passes between the return from ANAINP and the next call to this routine, this datum is already slightly outdated when it is retrieved at the beginning of the next pass of ANAINP, which may or may not matter. Each input value is immediately submitted to the digital low-pass filter routine in LOWPAS which corresponds to a first order analog low-pass (compare chapter 5.3.2.3.5). LOWPAS needs the previous data value of each analog channel which it deposited in the output array ANALOG; the contents of this array may, therefore, be read only but not modified. (This is no more true if ANAINP which calls LOWPAS is disabled altogether with the TEST flag.) Prior to calling LOWPAS, ANAINP prepares the A/D Converter board for the input of the next channel, programming the input multiplexer accordingly. The set-up time required by the multiplexer, i.e., the time which must pass before valid data can be submitted to the A/D Converter proper, is approximately equal to the execution time of LOWPAS. The A/D Converter hardware will therefore be ready

for the next step when LOWPAS has finished its job. ANAINP triggers the conversion proper when the A/D Converter board's hardware indicates that the board is ready; the routine waits in a loop until converted data are available. (The synchronization with the hardware is done with polling loops rather than with interrupts. This approach was preferable because each interrupt processed involves a considerable system overhead which takes several hundred microseconds. The A/D conversion is even faster than the processing of an interrupt, and the time required by the hardware for channel switching is used within ANAINP for the low-pass routine call.) Emergency timeouts were provided for either loop in order to avoid a total blockage of the system if the A/D Converter does not respond properly. (It turned out that the system <u>is</u> blocked, though, if no A/D Converter board is installed, and ANAINP is <u>not</u> disabled with TEST.)

5.3.2.3.3 The Relay Controller Routine MOTDIR

This assembly language routine provides the input from and the output to the digital (relay) interface. It has the following tasks:

(1) Provide output to the Controller Selection relay which must not be energized in Monitoring mode 0, and energized if the CGCS is in charge of the puller (i.e., in operation modes 1 through 4).

(2) Read the current status of the Motor Direction relays, and set the Motor Speed input values to zero if the corresponding motor is switched off.

(3) Check the sign of the Motor Speed output values, and provide the proper Motor Direction relay output.

(4) Determine the absolute value of the Motor Speed output for the D/A Converter.

A special approach had to be chosen for the Controller Selection relay output: Using just one output bit for turning on and off the relay would have been impeded by the fact that the status of the output port used may be undefined when the CGCS is not in charge of the controller computer. Furthermore, the PPI (Peripheral Parallel Interface) hardware comes up with all I/O lines in high impedance after a system reset, which would result in all relays either turned on or off. Therefore, three bits, namely bits 0 through 2 of one output byte, have to be set to defined values in order to actually permit con-

trol of the Controller Selection relay, and hence of all other relays: Bit 0 represents the Controller Selection output; it has to be high to switch control to the CGCS and to activate the Motor Direction relays, and it is low in Monitoring mode. In addition, bit 1 must be low, and bit 2, high, to enable the relays.

The Cambridge Motor Controller uses three relays for Motor Up/Clockwise, Motor Stop, and Motor Down/Counterclockwise, respectively. The CGCS is therefore connected to the puller by three relay control lines for each motor; these lines are energized by the Cambridge console if the analog circuitry is controlling the puller, and by the CGCS, if the CGCS is in charge. The status of the relay control lines is monitored by MOTDIR, and MOTDIR provides output to them when required. Since exactly one of the Motor Control relays must be energized for each channel, the status of the three lines may be represented by two bits:

Output:	Motor Status:	Speed Value:	Input:
0 0	Stop	0	0 0
1 0	Up/Clockwise	+	1 0
0 1	Down/Counterclockwise	-	0 1
1 1	Stop	0	0 0

MOTDIR uses two relays for the Motor Direction output whose contacts are wired to result in the above signals, i.e., the Stop line is energized if either no relay or both of them are on.

First, MOTDIR reads the current Motor Direction status, and resets the Motor Speed input value to zero if the Stop relay is on. This step prevents noise and offset errors within the analog circuitry from disturbing the Motor Speed output on the CGCS's console. The four times two Direction input bits are read and internally stored as one byte. Next, MOTDIR checks the Motor Speed setpoint values, and determines a relay setting according to the magnitude and sign of each setpoint. These four times two bits are also assembled in one byte. The input and output bytes are now "or"-ed, which sets all bits in the output byte which are set in one of the two input bytes, or in both. The two bits corresponding to one particular motor are forced to zero if a zero speed value was submitted as a setpoint. The resulting byte is output to the Motor Direction relays in any case. Actual output to the control lines is, however, only generated if the CGCS is in charge of the puller.

The chosen approach may appear unnecessarily complicated but it is, in fact, indispensable to guarantee valid Motor Direction signals. The combination of the output data with the previous input data has no effect if the direction of the setpoint speed is the same as the actual speed; the current motor direction will be maintained. The puller requires, however, a few tenths of a second in "Motor Stop" position if the rotation direction of a motor is to be reversed. This is automatically accomplished by the chosen approach: Both bits corresponding to one motor are set if the actual Motor Speed and the pertinent setpoint have different signs, which energizes the "Stop" control line. At the next pass of MOTDIR, one second later, two zero bits are input accordingly, and the Motor Direction output will be determined by the sign of the setpoint. A one second "Motor Stop" is therefore guaranteed in any case.

5.3.2.3.4 The Analog Data Output Routine ANAOPT

The Analog Data Output routine uses a similar approach as ANAINP for providing easily programmable output to random hardware channels of the D/A Converter board: The channel numbers are kept in an array ANOPAR whose size is not limited by ANAOPT. Output values are read from an array in the order in which they are stored; there is no limit to this array either. ANAOPT returns to the calling routine when a negative channel number is detected in the parameter array.

ANAOPT has to scale the data submitted to it by a factor of 8 since the D/A Converter board supports only a 12 bit unipolar data range. Round-off is provided according to the magnitude of the highest-order bit which has to be discarded. Negative output values are trapped and replaced by zero.

Incidentally, the (assembly language) routine ANAOPT is kept in the Data rather than in the Code area of the CGCS. This was necessary because ANAOPT "patches" its own program code according to the analog channel which is currently in use. This approach is, however, incompatible with the memory checking done by the Command Executor (compare chapter 5.3.1.4.8). (Although patching program code is legitimately considered a bad programming technique it was indispensable in this case because the 8085 processor used does not allow otherwise to access I/O port addresses which have been calculated before.)

5.3.2.3.5 The Low-Pass Filter Routine LOWPAS

The algorithm used by LOWPAS is very simple and efficient: With x_k, the current input value, and y_k and y_{k-1}, the current and the previous output values, respectively, LOWPAS calculates:

$$y_k = a \cdot x_k + b \cdot y_{k-1} \qquad (46),$$

with

$$a = 2^{-n} \qquad (47),$$

and

$$b = 1 - a = 1 - 2^{-n} \qquad (48),$$

where k and n are positive integers (0, 1, 2, ...). The restriction of eq. (47) permits a very fast evaluation of eq. (46). Eq. (48) guarantees an overall gain of 1 for constant (DC) signals. We can re-write eq. (46) to:

$$b \cdot (y_k - y_{k-1}) + (1 - b) \cdot y_k = a \cdot x_k \qquad (49)$$

Eq. (49) can be divided by T, the time interval between two runs of LOWPAS:

$$\frac{y_k - y_{k-1}}{T} + \frac{1 - b}{b \cdot T} \cdot y_k = \frac{a}{b \cdot T} \cdot x_k \qquad (50)$$

The difference in the left term in eq. (50) can be approximated by a differential, transforming eq. (50) to the differential equation:

$$\frac{\delta y}{\delta t} + \frac{1 - b}{b \cdot T} \cdot y = \frac{a}{b \cdot T} \cdot x \qquad (51)$$

This is evidently the response of a simple first-order (R-C) low-pass filter. Eq. (50) is an approximation, though, which is only valid for very slowly changing input values x_k. A more accurate analysis of the filter's frequency response must be based on the theory of digital filters: The complex frequency response $H(\Omega \cdot T)$, with

$$\Omega = 2 \cdot \pi \cdot f \qquad (52),$$

where f is the input signal frequency and T the time interval between two sampling points, can be obtained by a z-transformation of the filter's response to a single pulse with the

amplitude 1. It can easily be seen from eq. (46) that, for an input signal

$$\delta_k = \begin{cases} 1 & \text{for } k = 0 \\ 0 & \text{for } k > 0 \end{cases} \quad (53),$$

the output signal h_k will be:

$$\begin{array}{lccccccc} k = & 0 & 1 & 2 & 3 & & m \\ h_k = & a & a \cdot b & a \cdot b^2 & a \cdot b^3 & \ldots & a \cdot b^m \end{array} \quad (54)$$

With the definition of $H(\Omega \cdot T)$

$$H(\Omega \cdot T) = \sum_{k=0}^{\infty} h_k \cdot [\exp(j \cdot \Omega \cdot T)]^{-k} \quad (55),$$

and the summation formula for an infinite geometric series

$$1 + x + x^2 + x^3 + \ldots = \frac{1}{1 - x} \quad (56),$$

we can obtain the complex frequency response

$$H(\Omega \cdot T) = \frac{a}{1 - b \cdot \exp(-j \cdot \Omega \cdot T)} \quad (57).$$

Since we are not interested in the phase but only in the amplitude response, we derive the absolute value $|H(\Omega \cdot T)|$ from eq. (57):

$$|H(\Omega \cdot T)| = \frac{a}{(1 + b^2 - 2 \cdot b \cdot \cos(\Omega \cdot T))^{1/2}} \quad (58)$$

The cut-off frequency

$$\Omega_o = 2 \cdot \pi \cdot f_o \quad (59)$$

of a low-pass filter is defined as the point where the amplitude response drops to $1/\sqrt{2}$ of its DC value:

$$\frac{|H(\Omega_o \cdot T)|}{|H(0)|} = \frac{1}{\sqrt{2}} \quad (60)$$

With eqs. (58) and (60), we can write:

$$\cos(\Omega_o \cdot T) = 1 - \frac{(1 - b)^2}{2 \cdot b} \quad (61)$$

The cut-off frequency values listed in chapter 5.3.2.3.2 were obtained from eq. (61), with a sampling point interval T equal to 1 second.

6. CGCS Software Configuration

The CGCS consists of a number of FORTRAN and assembly language program source modules each of which holds one or several routines. These modules must first be converted into object machine code, which is done by a Compiler and Assembler program, respectively. The resulting object program files must be linked together by a special Linker utility which also resolves mutual references; the output of the Linker which still does not refer to absolute memory locations must be modified to do so by a Locate program. A special Configuration Module must be provided for the iRMX-80 operating system; this module may either be written in assembly language, or it can be prepared much more comfortably with a special Interactive Configuration Utility for iRMX-80, ICU-80. (All the development software mentioned is supplied by Intel.)

The actual configuration process is, however, much more complicated, due to the overlay structure chosen, and due to the fact that certain memory locations have to be "tied" together.

The configuration procedure starts with combining all assembly language and FORTRAN modules, respectively, which constitute the main body of the CGCS (i.e., the permanently resident code). These routines refer extensively to FORTRAN-iRMX-80 Interface, FORTRAN, and iRMX-80 library routines which are linked with the combined assembly language and FORTRAN code in the next step, together with two ICU-80-created modules. A dummy module, TRVMOD, included with the assembly language code, provides references which would be made by overlay routines otherwise; this permits to keep all support routines within the resident code. The following items are linked together in the order listed below:

```
FIRSTM.OBJ  (an auxiliary module required for the Program
            Code Integrity Check routines; compare chapter
            5.3.1.4.8).
The iRMX-80 Configuration Module.
All assembly language modules.
All FORTRAN modules.
FXUTIL.LIB  (compare chapter 5.2.4).
FIORMX.LIB  (compare chapter 5.2.2).
FXDISK.LIB  (compare chapter 5.2.3).
FRXMOD.LIB  (compare chapter 5.2.1).
FORTIO.LIB  (compare chapter 5.2.2.6).
RXIPUB.LIB  (compare chapter 3.4.2.3).
FP8231.LIB  (compare chapter 5.2.5).
LOD824.LIB  (iRMX-80 Loaded Systems library).
F80RUN.LIB  (FORTRAN-80 Runtime library).
F80NIO.LIB  (dummy FORTRAN I/O library).
```

FPEF.LIB (FORTRAN Intrinsic Functions library).
FPSFTX.LIB (FORTRAN software floating-point library).
F80NTH.LIB (dummy FORTRAN I/O library).
DFSDIR.LIB (iRMX-80 High-Level Disk I/O library).
DFSUNR.LIB (dummy iRMX-80 library).
An ICU-80-generated disk interface module.
TSK820.LIB (iRMX-80 Free Space Manager).
RMX824.LIB (main iRMX-80 library).
BOTUNR.LIB (dummy iRMX-80 library).
UNRSLV.LIB (dummy iRMX-80 library).
PLM80.LIB (integer arithmetics library).
LSTRAM.OBJ (an auxiliary module required for the Program Code Integrity Check routines; compare chapter 5.3.1.4.8).

(The various dummy libraries have essentially the purpose to "tie away" any unresolved references to unused external routines.) After this procedure, all references to external routines should be satisfied, with the exception of those referring to the routines which constitute Command Interpreter overlays. Despite these missing external references, the resident code is located to absolute memory addresses.

Next, the overlay routines (i.e., one or more program modules per overlay) are linked separately to one module for each overlay, combining them with the PUBLIC addresses of the resident CGCS code which were defined in the above locating step in order to satisfy their external references to routines which are part of the "body" of the CGCS. Since no overlay may directly refer to another overlay, no unresolved references may remain in the overlays, and the overlay code can be located to reside in the reserved overlay area.

The last phase, finally, entails linking the resident CGCS code to the PUBLIC start addresses of all Command Interpreter overlays, which satisfies the last yet open external references in the CGCS body. In addition, the Initialization code of the Command Interpreter which was prepared separately like an overlay (and which, indeed, resides in the memory area reserved for overlays) is linked to the resident CGCS code in its entirety. The resulting modules still contain a vast overhead of information which was required for linking and which is used by various debugging approaches. These references are not required for the execution of the program and would only unduly consume disk space and loading time; they are, therefore, stripped in the final step of software preparation.

A special technique is required to process the FORTRAN COMMON blocks properly: The ISIS-II LOCATE program places by default

COMMON blocks in an arbitrary order, at arbitrary memory locations. These locations must not only coincide for all overlays and for the program body to permit regular program operation, they must, at least in some cases, even be "tied" to locations used by assembly language programs. This entails that each of the 24 program modules which have to be prepared separately (i.e., the body of the CGCS, the Initialization code, the 21 Command Interpreter overlays, and the Data overlay) requires an explicit specification of the starting location of each COMMON block it uses when it is processed by LOCATE. Although this task can be automated by using appropriate SUBMIT (i.e., batch) files, it is very cumbersome to prepare these files, particularly because every additional COMMON block or every change of the size of a COMMON block requires the editing of 24 SUBMIT files. (Since Intel's LOCATE program prohibits the declaration of start addresses for COMMON blocks which are not present in the module currently processed, it is not possible to use generic SUBMIT files either.)

In order to facilitate the maintenance of the SUBMIT files which call LOCATE, a BASIC program, CSUBMT.BAS, was specially written. This program requires a source file (whose file name extension must be different from "CSD") which contains all information to be included in the SUBMIT file, except the yet undetermined addresses. The latter information must be supplied in a "dictionary file". CSUBMT.BAS replaces all lines in the source file whose first character is a "@" by the line in the dictionary file whose beginning is identical to the remainder of the line within the source file. Lines without a leading "@" remain unchanged. The program creates an output file with the same name as the source file but the extension "CSD" (which is Intel's reserved SUBMIT file extension). Since the address information is contained in the dictionary file only, changes of COMMON block locations require changes within one disk file (i.e., within the dictionary file) only; the probability of errors is thus greatly reduced. A small example shall show how CSUBMT.BAS works:

Suppose the LOCATE command should read:

```
LOCATE CZOV01.LNK TO CZOV01.LOC CODE(05400H) MEMORY(05C00H) &
STACKSIZE(0) &
/OVRLAY/ (0299AH) &
/OVLNM1/ (02994H) &
/COMMEX/ (028ACH) &
/COMMFL/ (028B6H) &
/SCALE/  (03571H) &
/SETPT0/ (02B9BH) &
MAP PRINT(CZOV01.LOM) PUBLICS SYMBOLS COLUMNS(3)
```

The pertaining source file for CSUBMT.BAS would be:

```
LOCATE CZOV01.LNK TO CZOV01.LOC CODE(05400H) MEMORY(05C00H) &
STACKSIZE(0) &
@/OVRLAY/
@/OVLNM1/
@/COMMEX/
@/COMMFL/
@/SCALE/
@/SETPT0/
MAP PRINT(CZOV01.LOM) PUBLICS SYMBOLS COLUMNS(3)
```

(There should not be any spaces following the contents of lines beginning with "@", i.e., these lines should be terminated immediately after the second "/" by a carriage--return.)

The above common block locations are extracted from the following dictionary file (which may, by the way, contain arbitrary comment lines):

```
   ...
/CNDCNT/ (028ABH) &
/COMMEX/ (028ACH) &
/COMMFL/ (028B6H) &
/CONLIM/ (028C0H) &
   ...
/MODE/   (02993H) &
/OVLNM1/ (02994H) &
/OVRLAY/ (0299AH) &
/PLOTAD/ (0299BH) &
   ...
/SECFLG/ (02B9AH) &
/SETPT0/ (02B9BH) &
/SETPT1/ (02BBCH) &
   ...
/LENGTH/ (0356FH) &
/SCALE/  (03571H) &
/AUXDIA/ (035B9H) &
   ...
```

The dictionary file may contain up to 100 replacement lines; only the items required for a particular SUBMIT file are selected.

A slightly different approach is chosen for the body of the CGCS program which accesses virtually all COMMON blocks. The start locations of all COMMON blocks which are accessed by FORTRAN modules only and which, therefore, need not be tied to locations defined by assembly language modules, can be defined

automatically by LOCATE; the addresses thus obtained must be "manually" copied to the dictionary file, though.

In addition to the main resident CGCS module CZOCHR.BIN and the 21 Command Interpreter overlay modules CZOV01 through CZOV21, plus the Data module CZOOVD, two more files are required on a CGCS system disk: The file CZOMEN holds a specially formatted Help menu which is displayed upon a HELP command (compare chapter 5.3.1.3.4); this file needs no special attention if a new system version is being generated, unless significant modifications of the command structure were made. A special treatment is, in contrast, required for the CZONAM file which holds the list of Variable addresses (compare chapter 4.7 and Appendices 11 and 12). This file must hold the current version code in its file name extension (CZONAM.V24 refers, for example, to version 2.4 of the CGCS), and it must be generated from a source file which may have required updating due to a possible shift of the addresses of some Variables because of software modifications. This source file is converted into the special CZONAM format (compare Appendix 12) by means of the auxiliary program CONVAD. CONVAD is designed to run under ISIS-II; it could also be executed under RXISIS-II but that will hardly be necessary.

The last step in preparing a work disk for a new CGCS version is, finally, up to the operator: All Macro command files which were used under a previous version and which are still required must be converted to the new system version using the facilities of the Macro Command Editor COMMED (compare chapter 7.2).

7. Supporting Programs for the CGCS

7.1 Data File Display Utility SHODAT

7.1.1 General Remarks

The program SHODAT allows to browse through Data files created by the Czochralski Growth Control System (CGCS) (compare chapter 4.6 and Appendix 12.4), and to dump the contents of selected records to a printer. Accesses to data records are permitted in a random mode, which allows to step back and forth through the growth data as required.

The CGCS creates three types of data records:

(1) Regular Data Output: Sets of all important system parameters - measured data, setpoints, controller outputs, calculated crystal diameter and length, and the four DEBUG Continuously channels, plus time and operation mode - are written to the disk in regular intervals as defined during the initialization of the Data file (with the CGCS's DATA or FILES command).

(2) Extra Data Output: Additional data sets with exactly the same format as the regular ones are written to the Data file upon each DUMP command, and at operation mode changes. (Note that the regular periodic data dumps to the Documentation output do not trigger output to the Data file.)

(3) Comments: Comments entered with the CGCS's COMMENT command are embedded between the data records in the Data file. Since the Data file is made up of equally sized records and most of a comment record is reserved for the storage of text, only a few system parameters - the time, the operation mode, and the crystal length - are recorded together with a comment.

SHODAT is designed to give a proper display of either record type. A CRT screen layout similar to the CGCS console screen is used for data display. Optional printer output is formatted in uniformly sized blocks five of which fit onto a printout page. A layout different from the one used for the CRT display had to be chosen in order to conserve paper.

Note: System parameters displayed on the CRT screen together with a comment line are, with the exception of the time, mode, and length grown information, leftovers from the record displayed immediately before. They may or may not have been

valid at the time the comment was recorded. There is no such ambiguity for the printer output of comments, though.

Time is recorded as an unsigned two-byte integer seconds count. Since the greatest unsigned integer which can be held in 16 bits is 65535 ($2^{16} - 1$), the system time "wraps around" to zero after 65536 seconds, i.e., after 18 hours, 12 minutes, and 16 seconds. The time information can be corrected by specifying a "Time Frame" number which indicates the multiple of 65536 seconds within which the system time lies. (The time display is, however, limited to 95 hours, 59 minutes and 59 seconds, after which time it starts again from zero.) SHODAT assumes that the first record of a Data file was recorded in Time Frame 1, i.e., at a system time between 0 and 18:20:11, and makes an intelligent guess at the correct time frame of later records. The first assumption may not be correct, though, if a Data file was started at an advanced stage of the growth run, and SHODAT's time evaluation may be incorrect if many extra data and comment records are inserted between the regular ones. Under worst case conditions, i.e., for a data record interval of 255 seconds, the minimum number of extra records which confuses SHODAT is 256; for shorter intervals, many more extra records are tolerated. Since even the minimum extra record count of 256 will hardly be attained under normal circumstances, the time displayed by SHODAT is fairly reliable if the proper starting time frame was chosen. Anyway, the time frame number can be interactively corrected by the operator with SHODAT's "T" command.

From release 2.0 of the CGCS software on, information is provided in the Data file about the display formats used for the four DEBUG Continuously output channels. This permits SHODAT to adapt its DEBUG output automatically to the proper format, or to blank the DEBUG data (but not the address information) if a DEBUG channel is inactive. The default format as supplied by the CGCS may be overridden, though, with SHODAT's "D" command; the new display format remains active until it is either replaced with a new one, or until the default format is restored with another "D" command. Data files generated with CGCS releases earlier than 2.0 can be displayed with SHODAT, too; SHODAT defaults to a four bytes hexadecimal DEBUG display format in this case (which provides the most complete information); the proper DEBUG Continuously format has to be entered manually.

Note: Parameters are recorded in unscaled two-byte integer format in the Data file, which entails that they must be scaled within SHODAT before they can be displayed. Modifications of the scaling factors within the CGCS may therefore result in invalid data displayed by SHODAT. The current

versions of the CGCS and SHODAT require that SHODAT is partly re-compiled to allow for scaling factor corrections. Since adjustments of the scaling factors within the CGCS also affect the proper operation of the Macro Command Editor and Display programs COMMED and READCM (compare chapter 7.2), input and output signals of the CGCS should be calibrated in any case by means of the analog input and output hardware, rather than by a change of their respective scaling factors.

7.1.2 Running SHODAT

SHODAT is available in an RXISIS-II and an ISIS-II version; it can therefore be executed either on the CGCS computer, or on an Intellec Series II Development System. It behaves identically on both systems but the program codes are different and not exchangeable. (SHODAT's performance under RXISIS-II is, however, superior to the ISIS-II version, due to the optimized console screen output and the high-speed floating-point arithmetics on the CGCS machine.) In either case, SHODAT is invoked by its name, without parameters.

Printer output is directed to the first serial port on an Intellec Development system (:TO:); this port must be properly initialized to the baud rate used by the printer before SHODAT is started.

SHODAT comes up with a sign-on message and a question referring to a "properly initialized and selected" printer. Any answer other than "Y(es)" (upper- or lowercase) disables SHODAT's "P(rint)" function and permits to run SHODAT safely on a system which is not connected to a printer. (SHODAT can be run as well with enabled Print functions on such a system. Any inadvertent "P" command would, however, cause the program to "freeze".)

Having requested the name of the Data file which is to be displayed, SHODAT reads its header record. Error messages are issued if the file does not have the proper format (or is no Data file altogether); otherwise, the header information and the file size are displayed. Answering the "OK?" question with "N(o)" permits to back out of SHODAT, while any other input leads on to the display of the first Data file record.

SHODAT displays the contents of the Data file on a screen similar to the CGCS's (compare Fig. 9), with the exception of the following differences:

(1) The actual time, the Macro command being executed, the number of parameters being "ramped", and the number of pending Conditional Macro commands are not available on the Data file. SHODAT displays, in the respective locations, the CGCS system version under which the Data file was created, the number of the current record in the Data file, the interval between Data records (in seconds), and the Time Frame number.

(2) One screen line (the same as in the CGCS) is permanently reserved for DEBUG output. The addresses linked to the four DEBUG Continuously channels are always displayed, even if the corresponding channels are inactive. The data area of the DEBUG display is, however, blanked for inactive channels if the Data file was created with CGCS release 2.0 or later. (An inactive channel maintains the address and data values last output on this channel.) A list of Variable names like the one in Appendix 11.3 can be used for translating the absolute hexadecimal addresses displayed by SHODAT into the pertinent Variable names.

(3) Immediately beneath the DEBUG display, the signals of the eight spare analog channels are displayed, scaled for a range from zero to ±100.

(4) A line of dashes separating the display of the spare analog channels from the command menu line is eventually replaced by the contents of a comment record.

(5) The last line but one on the screen holds a menu of the permitted commands.

The following command entries (in upper- or lowercase characters) are allowed in SHODAT; entries must be terminated with "Return":

<Return> ..... Advance to the next record.

<unsigned integer> Display specified record.

+ <integer> ..... Advance by specified number of records.

- <integer> ..... Step backwards by specified number of records.

In either of the above cases, output is limited to record numbers in the range from 1 to the total number of records in the file. Jumping to an "impossible" record number (e.g., 9999) displays the last record in the file.

P ..... Print record. Printer output generation halts SHODAT for a few seconds. Five records are printed to one output page.

T [<integer>] ..... Change Time Frame. A Time Frame number may be entered together with the "T" command; otherwise, it is explicitly requested. The Time Frame specified is taken as a base for subsequent time evaluation and remains effective until another "T" command is issued, or until a Time Frame number less than 1 would result, in which case the Time Frame number is reset to 1.

D ..... Specify a DEBUG Continuously display format. SHODAT requests the number of the DEBUG channel. If a Data file created under CGCS release 2.0 or later is under display, SHODAT asks "Override default Debug output mode?"; answering with "N(o)" restores the output mode to the one specified in the Data file. Modes entered with the "D" command remain in effect even if the mode defined by the Data file was changed subsequently; it requires another "D" command to modify them again.

E ..... Exit SHODAT.

7.2 Macro Command Editing and Displaying - Programs COMMED and READCM

7.2.1 General Remarks

The Macro Command Editor COMMED and the Macro Command File Display Program READCM support the Czochralski Growth Control System's Macro Command feature, i.e., the possibility to issue a series of timed Internal commands as defined by a disk file. Macro Commands are executed by reading and processing specially formatted encoded (tokenized) disk files which contain, in addition to the command code and the necessary parameters, a time tag. Each command recorded in a Macro Command file is executed with a particular time offset after the Macro Command was invoked which is specified in the Macro file (compare chapters 4.1.1 and 4.5).

There are, in general, two ways to create Macro Command files:

(1) By recording the commands actually issued during a growth run, and

(2) by using the Macro Command Editor COMMED which is totally independent from the Czochralski Growth Control System.

The Macro Command Editor COMMED and the Macro Command File Display Program READCM can be executed under RXISIS-II on the CGCS computer or, under ISIS-II, on an Intellec Series II Development System.

COMMED allows to generate Macro Command files, either from operator input on the console, from a genuine Macro Command file, or from a disk file whose format is compatible with files created with READCM. It permits to enter all commands which can be recorded on a Macro file, together with the command execution time, and it allows to edit old Macro Command files. COMMED generates a Macro file for the latest program version of the CGCS which resides on the specified target disk. (Macro Command files can only be generated on disks which hold a Variable name file CZONAM.Vmn, where m and n are integers representing the CGCS release version; compare chapter 6 and Appendix 12.1.)

READCM is a relatively simple program which permits to convert encoded Macro Command files into legible form; its output may be directed to a printer for documentation purposes, or to a disk file which, in turn, may be used as an input to COMMED.

Note: Values of primary CGCS parameters - diameter, heater temperature(s), motor speeds, and power limit - which are

specified with SET or CHANGE commands are stored in the Macro command file with the internal two-byte integer representation used by the CGCS. It is essential that the scaling factors used by COMMED and READCM for these parameters are identical to those applied by the CGCS. The inconsiderate modification of scaling factors within the CGCS will, therefore, result in discrepancies between the commands as seen by COMMED and READCM, and as executed by the CGCS!

7.2.2 The Macro Command File Editor COMMED

COMMED is primarily designed for interactive use although it can be run under the control of a SUBMIT file (under ISIS-II only). The Macro Command Editor uses all available memory as a buffer for building and chronologically sorting the Macro Command file to be edited; hence, the number of Internal commands within a Macro Command file is limited. (The limit lies between 600 and 800 commands, depending on the environment and the COMMED version used, which is probably more than sufficient for all practical purposes. The maximum permitted number of commands is displayed by COMMED as part of its sign-on message.)

COMMED can either be used to create Macro files from scratch, or to edit existing files. In the first case, input is exclusively retrieved from the keyboard, in the second, from an existing Macro file. COMMED permits to read genuine Macro Command files, and Macro List files which may have been created during a previous pass of COMMED, or with READCM. Macro Command files which are used as an input for COMMED are first automatically translated into List file format; COMMED uses the Variable name file CZONAM for the CGCS version under which the source Macro file was generated in order to convert absolute memory references into symbolic Variable names. (COMMED terminates immediately if the correct CZONAM file is not found on the disk which contains the source Macro.) The output file created by COMMED may be stored under the same name as the source file if a Macro Command file is used as a source; COMMED issues a warning in this case, though, that an existing file is to be overwritten. No file name extension need be specified if Macro Command file names are entered; COMMED appends ".CMD" automatically. COMMED searches the disk on which the output Macro file is to be stored for the latest version of the Variable name file CZONAM; it exits with an error message if no such file is found. Since COMMED creates its output files always for the latest CGCS version available on the target disk but allows to read Macro files for arbitrary CGCS releases if the proper CZONAM files are found on

the source disk, it automatically translates Macro files for use with the latest CGCS version. (Since the addresses of Variables may have changed due to modifications of the CGCS software, the CGCS prohibits the execution of commands referring to absolute addresses if a Macro file was created under or for a different CGCS version; compare chapter 4.5.)

Before COMMED enters the actual file editing sequence, it permits the definition of a Macro List file to which it will direct a listing of the commands within the Macro file in single line format (compare chapter 4.3.2). The List output may be directed to a disk file, or to a printer (if the proper device - ":LP:" or ":TO:" - is specified). Aside from documentation purposes, such a List file can be used as an input for future COMMED runs. It is not possible, though, to redirect the List output to a List file which is currently used for input.

In order to optimize the efficiency of COMMED for the various applications for which it may be used - creation of a new Macro Command file, editing of an existing file, translation of a Macro file for a new CGCS version, or generation of a List file only (for which purpose READCM is the better choice, though) -, the actual command editing features of COMMED can optionally be disabled. If editing was declined by the user, COMMED simply processes its input file command line by command line, and requires operator input only if it detects a possible command error. This operation mode is obviously best suited for the translation of Macro Command files; it should be noted, though, that only references to Variables which are defined in the source and in the target CGCS version can be resolved automatically. COMMED skips the current command if it does not find the name of a Variable in the target CZONAM file; commands referring to locations which were specified by their absolute addresses in the source file (e.g., in a DEBUG or PLOT command) will be copied into the target file unmodified. Since potentially disastrous problems might arise from referring to an absolute address which may be incorrect in the new CGCS version, the use of absolute addresses within Macro Command files is strongly discouraged.

If editing is activated, each command line of the input file is displayed (if there is an input file), and the user may accept, delete, or replace it, or insert a command in front of it. The execution time as defined by the source file is displayed together with the command line; it cannot be edited, but an optional offset may be subtracted from the relative time of each command. This feature permits to "break" a lengthy Command Output file created by the CGCS into a number of Macro Command files each of which starts at time 0 (or

approximately at time 0). COMMED discards all commands which would get a time tag of less than 1 after it subtracted the time offset. (The default time offset is, of course, 0.)

The following commands may be legally entered in COMMED; all other CGCS commands will result in a (non-fatal) error:

CHANGE
CLEAR
DEBUG Continuously
DEBUG Modify
DEBUG Off
DEBUG Resume
DEBUG Suspend
END
HELP or ?
IF
MODE
PLOT
RESET
SET
Macro Command names

Command entry is equivalent to the CGCS (compare chapters 4.3.2 and 4.3.3) (with the exception that a few commands may be entered in one line with COMMED which require several input lines in the CGCS, e.g.,the MODE command). Each command must be preceded by the relative time of its execution; COMMED proposes a value which may be accepted (with "Return") or replaced by an arbitrary positive execution time value. Commands need not be entered in chronological order; once a command was entered, though, it cannot be modified or removed during the current pass of COMMED. In general, the command entry dialogue of the CGCS is duplicated by COMMED, except in cases where this is not possible (since COMMED obviously cannot know, e.g., the actual value of a parameter to be SET or CHANGEd at the time the Macro Command is executed). COMMED checks for the existence of Macro command files, though, whose names are specified in Conditional or unconditional Macro commands, and issues a warning if no such file was found on the target disk.

After command entry was terminated (with "<EOF>" instead of a command execution time), COMMED sorts all commands chronologically, and displays them on the console and writes them to the List file in their final sequence. A count of errors (which generally result in the cancellation of the command affected) is displayed, and COMMED allows the user to continue with editing another file without having to re-load COMMED (which is rather time-consuming, due to COMMED's large size).

7.2.3 The Macro Command File Display Utility READCM

READCM provides a subset of the functions offered by COMMED. Its purpose is restricted to the conversion of Macro Command files into Macro List files; requiring less code and setup operations, READCM is faster and easier to use than COMMED.

Similar to COMMED, READCM scans the disk which holds the source Macro Command file for the proper Variable name file CZONAM. Unlike COMMED, however, it does not quit its operation if no such file exists but simply displays the absolute hexadecimal addresses of all non-primary parameters, even for commands which operate on Variables only (i.e., CHANGE, CLEAR, DEBUG, IF, PLOT, and SET). The resulting output file cannot be used with COMMED in this case, but it contains still the full information of the source file.

The Macro List file generated by READCM may be routed to a disk file or directly to a printer; it is displayed on the console screen only, if no output file is specified. Each line of the List file holds one command (in single-line format; compare chapter 4.3.2), preceded by the time (in seconds) relative to the start of the Macro command. (COMMED uses exactly the same format for its List output.) Macro List files created by READCM or COMMED can be further processed with COMMED; they may also be edited with any suitable text editor (e.g., Intel's CREDIT; compare chapter 3.4.1.2.1) prior to be submitted to COMMED again.

Appendix 1: Additional Documentation

iRMX-80™ User's Guide; Intel Corporation 1979, 1980; Manual Order No 9800522-05:
General information about iRMX-80.

iSBC 80-24 Single Board Computer Hardware Reference Manual; Intel Corporation, 1980; Manual Order No 142648-001:
Hardware reference manual.

iSBX 331 Fixed/ Floating-Point Math Multimodule Board Hardware Reference Manual; Intel Corporation, 1980; Manual Order No 142668-001:
Hardware reference manual.

iSBC 016A/032A/064A/028A/056A RAM Board Hardware Reference Manual; Intel Corporation, 1981; Manual Order No 143572-001:
Hardware reference manual.

iSBC 517 I/O Expansion Board Hardware Reference Manual; Intel Corporation, 1977, 1979; Manual Order No 9800388-01:
Hardware reference manual.

iSBC 204 Flexible Diskette Controller Hardware Reference Manual; Intel Corporation, 1978; Manual Order No 9800568A:
Hardware reference manual.

User Manual for DT772 Series Analog Input Systems for MULTIBUS Computers; Data Translation, 1984; Document UM-02829-A:
Hardware reference manual.

The Alternative Loader Task - Library ROLOAD.LIB; Karl Riedling, September 1984:
Information within this documentation complements and replaces information in the iRMX-80™ User's Guide. *)

The Alternative Terminal Handler - Library ATHxxx.LIB; Karl Riedling, February 1985:
Information within this documentation complements and replaces information in the iRMX-80™ User's Guide. *)

RXISIS-II User's Guide; Karl Riedling, April 1987:
General information about RXISIS-II and its supporting routines, the RXISIS-II Monitor, and the RXISIS-II Confidence Test. Short overview over utility software available under RXISIS-II. *)

ISIS-II User's Guide; Intel Corporation, various issues and order numbers:
Documentation of Intel supplied utility software which is also available under RXISIS-II.

Additional System Programs for Intel Development Systems; Karl Riedling, March 1981:
Documentation of additional utility routines; all programs listed are compatible with RXISIS-II. *)

Fortran - RMX-80 Interface Program Package; Karl Riedling, May 1987 (Issue 4):
Extensive documentation of all Interface routines used in the CGCS, containing also discussions of various programming approaches and of the system configuration. *)

Additional Fortran Numeric Routines; Karl Riedling, 1985:
Documentation of alternative Fortran floating-point system routines which use the 8231 Numeric Processor. *)

Czochralski GaAs Crystal Growth Controller - Short Reference; Karl Riedling, December 1986 (Issue 4):
User's reference manual for the CGCS. *)

Czochralski GaAs Crystal Growth Controller - Operator's Manual; Karl Riedling, December 1986:
Operation guide for RXISIS-II and the CGCS. (Subset of the Short Reference and Digital Controller Emergency Procedures manuals.)

Czochralski Growth Control System - Digital Controller Emergency Procedures; Karl Riedling, December 1986:
Procedures for emergencies caused by the CGCS hardware or software. (Part of the Operator's Manual.)

Czochralski Growth Control System Macro Command Editor COMMED; Karl Riedling, April 1986:
User's reference manual for the Macro Command Editor programs COMMED and READCM. *)

Program SHODAT - Short Reference; Karl Riedling, May 1986 (Issue 2):
User's reference manual for the Data file display utility SHODAT. *)

*) Essential parts of these documentations have been merged into the present Czochralski Growth Control System Reference Manual.

Appendix 2: Hardware Setup and Testing

This Appendix is structured as follows:

A: Defines an action in the setup procedure.
R: Specifies the result to be expected.
F: Suggests failure mechanisms if the system failed to produce the results under R:.

A: Install the four RXISIS-II EPROMs in the correct order on the iSBC 80-24 board. Connect a CRT terminal to the RS232 interface on the iSBC 80-24 board, and a printer to the iSBC 517's RS232 connector. Switch the terminal and the printer on, set the printer to on-line, and wait for the terminal to warm up. Insert a write-enabled scratch disk in each disk drive. (Data on these disks will be destroyed during the confidence test!) Switch the disk drives and subsequently the computer on.

R: A flashing display of zeros and ones should appear in the top left corner of the console CRT, indicating the Memory Confidence Test, and three "beeps" should be issued, one, fractions of a second after power-on, and, after an interval of several seconds, two more within fractions of a second. After the third "beep", the display of zeros and ones should be overwritten by the sign-on message of the RXISIS Monitor.

F: No display at all, processor not running - check hardware.

No display, processor running - check RS232 connection and terminal.

Processor runs and halts with display of zeros and ones - ROM or RAM failure - check the Confidence Test documentation in chapter 3.3.2. You can restart the system with a Reset; the Confidence Test will be skipped, and you can enter the Monitor for inspection of the memory. (The memory page in which a RAM error was detected is indicated by the binary data displayed; the actual location can be determined by inspecting the memory with the Monitor's "D" command. It is the location where the RAM contents change to a different value.)

A: Press the space bar to enter the Monitor, and invoke the Confidence Test with the "Z" command. Run the Confidence Test (see chapter 3.3.2). (If you want to test I/O ports there is a more comfortable routine in the Monitor. You

may therefore skip the Confidence Test's I/O Port Test.) The Confidence Test returns to the Monitor, and you may invoke it repeatedly.

R: You should obtain an error-free operation through the entire Confidence Test. (Overrun errors detected during the CRT Console Test are due to a too fast entry on the keyboard. You cannot blame the system for them.)

F: System halts within the Memory Test - see above.

Parity errors during CRT Console Test - check terminal.

Printer does not print - check printer and its interface.

Errors during Floppy Disk Test:

NOTE: The test routine and the ROM resident iRMX-80 Disk File System routines have been configured for Shugart 800/801 drives. Different drives need not necessarily work properly with the software supplied.

"Drive not ready" - check jumper configuration on the disk drive.

Any other disk error - possibly a problem with the drive and/or the scratch disk used.

A: Enter the Monitor again. Try a few Monitor commands (see chapter 3.3.1). Command "P1" to duplicate the console output to the printer. Command "D4000,8000" (or any other memory range which is large enough to make the output generated by the Monitor completely fill the printer's internal buffer). Check the printer output for missing data.

R: No data should be missing even if the output on the console (which is linked to the printer output) is temporarily halted. If the printer buffer is so large that emptying it requires more than approximately 12 seconds, a message "PRINTER TIMEOUT" may be issued; the printer output is deactivated in this case. Set the printer buffer to a smaller size if this happens and if it is feasible. (A "PRINTER TIMEOUT" condition at this stage does not constitute a problem for the operation of the CGCS, though.)

F: Missing data indicates a problem with the handshaking between the printer and its interface. Check the cabling and the printer configuration.

A: Keep the printer in on-line mode. Insert the disk labeled "I/O AND 8231 APU TEST" in Drive 0. Enter "Q" (for "quit") while in the Monitor, and answer with "Y(es)".

R: The disk will be accessed, and after a few seconds of loading, the CRT screen will be erased (not necessarily if the CRT terminal uses control codes different from those for which the software was configured, which imposes, however, only a minor problem). A prompt line "Enter two floating-point numbers:" will appear. The CPU will enter a "Halt" state (there are no CPU "Halts" in the Monitor and in the Confidence Test, except in the case of a memory error detected).

F: The operation of the disk-loaded test program is based upon iRMX-80 and therefore on interrupts, in contrast to the Confidence Test and Monitor which operate with polling loops. All functions which did work under the Confidence Test and/or the Monitor but do not work under iRMX-80 indicate a hardware problem related to the interrupts.

A: Type in two arbitrary numbers, followed by "Return" after the second number.

R: The program performs a number of arithmetic operations with the standard floating-point algorithms of FORTRAN, and with the 8231 APU on the iSBX 331 Multimodule Board. The results of both operations are compared, and the absolute and relative differences are displayed. The results obtained from either source should match one another reasonably; except in the case of singularities, the relative errors ought to lie in the order of 1.E-6 or less. The output on the console CRT screen is duplicated on the printer.

F: In case of arithmetic errors or of a catastrophic program malfunction, check the iSBX 331 board. If "PRINTER NOT READY" messages are output although the printer is ready and worked properly under the Monitor, the most likely reason is that the Printer USART interrupts are not properly recognized.

A: You can disable the printer output at any time with "Cntl-V". A second "Cntl-V" enables it again.

Press the "Break" key on the console terminal or the "Interrupt" switch on the cardcage. In either case, a Monitor sign-on message is displayed. You can return to the test program with the Monitor's "G(o)" command.

You can invoke the iRMX-80 Debugger with "Cntl-C" while in the iRMX-80-based test program (a full active Debugger is included with the test routines). A return from the Debugger can be effected with "Q" or with "Cntl-A".

Finally, you can replace the I/O Test disk by an RXISIS-II or CGCS system disk, enter the Monitor (via a "Break" or an RST5.5 interrupt) and load RXISIS-II (with the Monitor's "Q(uit)" command). Alternatively, you can bootload RXISIS-II with a system reset followed by a "Return". Try to display the disk directory (with the "DIR" command), to format a blank disk in drive 1 (with "FORMAT :F1:<diskname>"), and to copy the contents of the system disk to the disk in drive 1 (with "CPYDSK :F0: TO :F1:"). No error messages should be displayed at this stage any more.

Appendix 3: Operating System Memory Allocation

ROM Resident Program Code:

Restart Vectors:	ROM		0000 - 003F
RXISIS-II MEMTOP:	ROM	BK1	0004 - 0005
Checksums:	ROM		0020 - 0023
Confidence Test:	ROM	BK0	0040 - 0FFF
Mcnitor:	ROM	BK0	1000 - 1FF7
iRMX-80 Nucleus:	ROM	BK1	0040 - 1FF7
Monitor Return Code:	ROM		1FF8 - 1FFF

Data Areas for ROM Resident Code:

Confidence Test/Monitor:	RAM	2000 - 200F
Vector to R?RST5HD:	RAM	2010 - 2012
iRMX-80 Nucleus:	RAM	2018 - 27CF
Cursor Positioning:	RAM	27D0 - 27FF
ROM Linkage Version	RAM	FEAE - FEAF
Controller Buffer	RAM	FEB0 - FEFF
Loader Buffer RQLBUF:	RAM	FF00 - FFFF

Program and Data Area for Applications:

Available RAM:	RAM	2800 - FEAD

RXISIS-II:

RXISIS-II Subroutines + Data	RAM	C800 - F6FF
RXISIS-II Internal Buffer	RAM	F700 - F7FF
RXISIS-II Main Body	RAM	F800 - FEAB
RXISIS-II Version Code	RAM	FEAC - FEAD
RXISIS.CLI	RAM	3002 - 3FFF
RXISIS-II Linkage Version	RAM	3000 - 3001
BOTLOD (System Bootloader)	RAM	3400 - 3402
iRMX-80 Debugger under RXISIS-II	RAM	9300 - C7FF
Application Program Area	RAM	2800 - C7FF
(with Debugger:	RAM	2800 - 92FF)

iRMX-80 BASIC:

BASIC	RAM	2800 - AFFF
Workspace	RAM	B000 - FEAD
ROM Linkage Version	RAM	FEAE - FEAF

Appendix 3: Operating System Memory Allocation

Entry Points in Both ROM Banks:

0000H	Entry Point for System Reset (RST 0)
0008H	Entry Point for RXISIS Re-Boot (RST 1)
000FH	Entry Point for Excess Returns from Routines Executed Under the Monitor
0010H	Main Entry Point to the Monitor (RST 2)
0018H	Auxiliary Monitor Entry Point (RST 3)
	Monitor Interrupt: RST 5.5
1FF8H	Return from Monitor Sequence

Entry Points in ROM Bank #0:

0040H	Coldstart Confidence Test
0043H	Extended Confidence Test
1000H	Regular Monitor Entry Point
1003H	Auxiliary Entry Point
1006H	Monitor Initialization Routine
1009H	Disk I/O Error Message Output

Entry Points in ROM Bank #1:

0040H	Vector to ISIS-II Emulator Routines
0043H	Vector to RXISIS-II Start and Bootloader Routines
0046H	Vector to Bootloader for Modules Other than RXISIS B+C must hold the address of the file name block of the file to be bootloaded
0080H	iRMX-80 Nucleus Start Module

Entry Points in RAM:

2010H	Entry Point from RST5 - can be overwritten by jump to R?RST5HD if the debugger is included
27D0H	Vector to Cursor Positioning Routine
27D3H	Vector to Line Clearing Routine

Entry Points to ISIS-II Emulator RXISIS-II:

F800H	ISIS
F803H	CI
F806H	Not Implemented (RI)
F809H	CO
F80CH	Not Implemented (PO)
F80FH	LO

F812H	CSTS
F815H	Not Implemented (IOCHK)
F818H	Not Implemented (IOSET)
F81BH	MEMCHK
F81EH	Not Implemented (IODEF)
F826H	Not Implemented (UI)
F829H	Not Implemented (UO)
F82CH	Not Implemented (UPPS)

Entry Points to Command Line Interpreter:

3002H	Bootloader for Non-RXISIS-II Systems

Monitor/Confidence Test Memory Locations:

2000H	Coldstart Check Byte
2001H	Monitor Status Byte
2002H	Breakpoint #1 Address
2004H	Breakpoint #1 Data
2005H	Breakpoint #2 Address
2007H	Breakpoint #2 Data
2008H	Exit Code Pointer
200AH	Space for "IN" Machine Code Instruction Byte
200BH	Input Port Address
200CH	Space for "RET" Machine Code Instruction Byte
200DH	Space for "OUT" Machine Code Instruction Byte
200EH	Output Port Address
200FH	Space for "RET" Machine Code Instruction Byte
27D6H	Cursor Up Code
27D8H	Cursor Down Code
27DAH	Cursor Left Code
27DCH	Cursor Right Code
27DEH	Cursor Home Code
27E0H	Clear Entire Screen Code
27E2H	Clear Line Code
FFF2H	Storage of PSW
FFF4H	B+C
FFF6H	D+E
FFF8H	H+L
FFFAH	PC
FFFCH	Stackpointer

Appendix 4: Disk Error Codes

RXISIS-II and the CGCS return a numeric error code in the case of a disk error. The error codes are the same in either environment for a given error condition. Although some error messages are trapped by application programs (under RXISIS-II) or by the CGCS, and replaced by more detailed message text, many errors are displayed by the generic error message generation routines which provide the error code only, without an explanation. The CGCS returns, for example, a message

***** DISK ERROR xxx yy (TASK tsknam, LOC hexl) *****

which is accompanied by a "beep". In the above message, "xxx" is replaced by the major, and "yy", by the minor error codes; "tsknam" stands for the name of the task which detected the error, and "hexl" represents the absolute program code address where the error was recognized.

The task name displayed with the disk error message indicates which CGCS file was involved in the error:

General System Operations:
    RXIROM - Overlay or auxiliary file handling.

Macro Command File:
    RXIROM - (Conditional) Macro call from console.
    CMMDEX - (Conditional) Macro call from console or Macro file.
    CMFINP - Macro command execution.

Print File:
    RXIROM - At all times.
    CMMDEX - (Error) message output.
    CMFINP - (Error) message output.
    DIACNT - (Error) message output.

Data File:
    RXIROM - During opening and closing and upon a COMMENT command.
    DSKOUT - During regular operation.

Control Output File:
    RXIROM - During opening and closing.
    CMFOUT - During regular operation.

The following error codes are returned by RXISIS-II and by the CGCS:

2	Invalid file number
3	Attempt to open more than 6 files simultaneously *)
4	Illegal file name
5	Illegal device name
6	Attempt to write to a file open for input
7	Disk is full +)
8	Attempt to read from a file open for output
9	Disk directory is full
10	Different disks in RENAME call
11	File name is already in use
12	File is already open
13	No such file
14	Attempt to write to a write protected file
15	Attempt to load into protected memory area +)
16	Incorrect object program format +)
17	Attempt to access a non-disk file
18	Unrecognized message type or system call +)
19	Attempt to seek on a non-disk file
20	Attempt to seek in front of beginning of a file
22	Illegal access parameter in OPEN call
24	Disk I/O (hardware) error +)
26	Illegal attribute parameter in ATTRIB call
27	Illegal mode parameter in SEEK call
28	Missing file name extension *)
29	End of console file
30	Disk drive not ready +)
31	Attempt to seek on a file open for output
32	Attempt to delete an open file
33	Illegal system call parameter @)
35	Attempt to seek past end of file open for input
40	Request sent to wrong exchange
41	Insufficient free memory to open file
42	Drive not in configuration table
43	Drive timeout
44	Seek request with seek not present in system #)
100	Disk overlay does not match ROM system version *) +)
101	Missing entry point in disk overlay *) +)
102	Illegal system call *) +)
120	Insufficient memory to open new file #)
121	Attempt to load a main program #)
218	Unallocated disk file block prior to EOF +)

@) ISIS-II only
*) RXISIS-II only
#) CGCS only
+) Fatal error under RXISIS-II

Minor error code information is only displayed in case of an error 24 (Disk I/O error):

01	Deleted record
02	Cyclic redundancy check error (data field)
03	Invalid address mark
04	Seek error
08	Address error
0A	Cyclic redundancy check error (ID field)
0E	No address mark
0F	Incorrect data address mark
10	Data overrun or underrun
20	Disk is write protected
40	Write error
80	Not ready

Appendix 5: Command Line Editing and Control Characters under RXISIS-II and the CGCS

(1) Line Termination Codes:

The following codes terminate an input line and advance the input buffer to the routine requesting input. In general, the termination characters are appended to the input data unless there is no more enough room in the buffer.

CR — Carriage Return: Converted to a CR-LF pair and written to the buffer and echoed as CR-LF.

LF — Line Feed: Treated identically to Carriage Return.

ESC — Escape: Appended to the buffer, echoed as "$"+ CR-LF (no "$" if entered in type-ahead mode). The RXISIS-II Command Line Interpreter and the RXISIS-II applications listed in chapter 3.4 and in Appendix 6 interpret Escape as a line termination and input character (analogous to Carriage Return), in contrast to the CGCS and the CGCS-related utilities (SHODAT, COMMED, READCM) which are based upon the I/O Interface Routines discussed in chapter 5.2.2; these routines use Escape as an input line clearing command (similar but not totally equivalent to Cntl-X).

Cntl-Z — Control-Z: Deletes all buffer contents, transmits an empty buffer. Echoed as a CR-LF pair. In type-ahead mode, Cntl-Z may be used to delete the last, yet unterminated, line entered, without affecting the contents of preceding input lines. (Due to system timing problems which could be overcome only with a great expenditure of code and/or processing time, an input line entered into the type-ahead buffer immediately after one or more Cntl-Z characters may not be echoed although it is regularly advanced to the task requesting input.) Control-Z is interpreted as a program termination code by some auxiliary programs (e.g., the Macro Command Editor) running under RXISIS-II.

(2) Line Editing Codes:

RO Rubout: Deletes the last character in the input buffer and on the screen. (It is, however, impossible to remove a character in the last column of the CRT screen from the display if the terminal used does not permit a "scroll-back" of the cursor from the leftmost position of a line into the last position of the preceding line. Nevertheless, the erased character is removed from the input buffer; a correct display can be obtained with Cntl-R.)

Cntl-X Control-X: Deletes the buffer and the type-ahead buffer completely. Appends a "#" to the input line echo and advances to the next line on the screen (not in type-ahead mode).

(3) Miscellaneous Control Codes:

Cntl-P Control-P: The character following Cntl-P is input literally even if it is a control character.

Cntl-R Control-R: Restores the input line echo on the console CRT. No visible effect if input line does not extend over two physical CRT screen lines. Can be used if characters were deleted in an input line extending over two CRT lines and the cursor did not move up to the upper echo line.

Cntl-S Control-S: Suspends regular console output. The tasks requesting regular output are halted. Does not affect output sent to the RQALRM exchange. Suspending output already suspended has no effect.

Cntl-Q Control-Q: Resumes output suspended with Cntl-S. Resuming output not suspended has no effect.

Cntl-O Control-O: Regular output is deleted if Cntl-O was entered. The routines or tasks generating output keep running; their output is lost. Regular output can be restored if Cntl-O is entered a second time.

Cntl-E Control-E: Suspends printer output. Tasks requesting printer output are halted. Suspending printer output which is already suspended has no effect.

Appendix 5: Command Line Editing and Control Characters

Cntl-F Control-F: Resumes printer output suspended with Cntl-E. Resuming output not suspended has no effect.

Cntl-V Control-V: Printer output is deleted if Cntl-V was entered. Tasks requesting printer output keep running; their output is lost. Printer output can be resumed if Cntl-V is entered a second time.

Cntl-C Control-C: This control is only effective if the Debug Enable Flag RQDBEN is set, if the regular input exchange RQINPX is active, and if a request message waits at RQDBUG. In this case, all console input is directed to the request messages waiting at RQDBUG, and RQINPX is no more serviced. In addition, a message is sent to the exchange RQWAKE unless there is a message already waiting there. The regular input mode is restored and the Mode Changed Flag RQTHMC set to 0FFH if a message of LASTRDTYPE (10) is sent to RQDBUG. The type-ahead buffer is cleared. (Control-C is used by the iRMX-80 Debugger, and by iRMX-80 BASIC. It has no effect whatsoever in the CGCS.)

Cntl-A Control-A: Cancels the effect of Cntl-C; all input is directed to the regular input exchange RQINPX. Cntl-A is only active if RQDBEN is set and RQINPX was not serviced (i.e., in Debug mode). The Mode Changed Flag RQTHMC is set to 0FFH, and the type-ahead buffer is cleared. (Control-A is used in conjunction with the iRMX-80 Debugger only; it has no effect whatsoever in the CGCS.)

Appendix 6: Utility Programs Under RXISIS-II

In addition to the standard ISIS-II utility programs (compare "ISIS-II User's Guide"), the following utilities are available on the CGCS computer under RXISIS-II:

Appendix 6.1: File Attribute Modification Utility ATTSET

This program modifies the file attributes either of all files on a disk, or of a number of files specified by a selection list. The latter feature constitutes its major advantage over the ISIS-II system program "ATTRIB". All files handled and their attributes after the execution of "ATTSET" are listed on the console. Upon program termination, the total number of files processed is displayed and, if applicable, a warning that files specified by the selection file have not been found.

PROGRAM CALL: ATTSET :F<n>: [switches]

A valid disk drive must be explicitly specified (even for :F0:).

[switches] may be any arbitrary optional sequence of one or more of the following commands in upper- or lowercase characters, separated by at least one space:

C <select-file>: Only the files listed in <select-file> are processed. (Note that all files are processed if the "C" switch is not specified.) <select-file> may be any valid ISIS-II disk file or input device name. The selection file may contain any number of file names without drive specifications in arbitrary order. The file names must be separated by one or more of the following characters:

* Spaces
* Carriage-return - line feed pairs
* "Escape" characters
* TABs
* Any other control characters

If a file listed in the selection file is not found on the disk, an appropriate error message is output; execution is continued for the next submitted file name. No "wild card" filenames ("*.SRC") are permitted.

I{0|1}: This switch resets (0) or sets (1) the "invisible" attribute of the specified files. Either "0" or "1" must immediately follow the "I".

S{0|1}: This switch resets (0) or sets (1) the "system" attribute of the specified files. Either "0" or "1" must immediately follow the "S".

W{0|1}: This switch resets (0) or sets (1) the "write protect" attribute of the specified files. Either "0" or "1" must immediately follow the "W".

REMARKS:

Attributes not specified in the program call are not affected. If none of the attribute switches (I, S, or W) has been entered, the current attributes are displayed without being modified.

"ATTSET" does not permit any modification of the "format" ("F") attribute, nor does it allow any access to a file with the "F" attribute set. This feature was provided in order to protect the ISIS-II programs from accidental deletion.

The names of the files to be processed may be entered directly on the console if "ATTSET" is invoked with the switch "C:CI:". This is particularly advantageous if only a few files with totally different names have to be processed (and if therefore there is no possibility of using the "wild card" feature of "ATTRIB"). Note: The execution of "ATTSET" is started only after the file names input on the console were actually entered (by pressing "Return" or "Escape"). Several file names may be specified within one input line, and the number of input lines is unlimited. The execution must be terminated by entering "CNTL-Z" after all files were processed.

A selection disk file may be generated from scratch with "CREDIT" or "CREATE". It is probably more convenient to generate a suitably formatted copy of the disk directory by a "DIRFIL" call and to edit this file with CREDIT if a large number of the files on one disk are to be processed. The same file can be used for all file selecting programs within this package ("ATTSET", "CMPDSK", and "CPYDSK"), no matter on which drive the disk to be processed is mounted. (This is why the selection file must not contain any drive references.)

A multiple specification of a file name within the selection file does not matter.

In the case of multiple but contradictive switch definitions within one program call, the last definition entered is considered valid. Multiple entries of the "C" switch, however, cause a fatal command error.

Appendix 6.2: Disk Comparison Utility CMPDSK

This program compares either all files on a disk or a number of files specified by a selection list to the corresponding files with the same names on a second disk. The results of the comparison (performed on a byte-by-byte basis) are displayed on the console for each file pair. The program execution is terminated by summarizing the total numbers of identical file pairs, of different file pairs, and of files contained on the first but not on the second disk. If applicable, a warning is issued that files specified by the selection file have not been found on the first disk. The comparison algorithms and the messages are identical to those used within "COMP"; for further information, see the corresponding paragraph.

PROGRAM CALL: CMPDSK :F<m>: TO :F<n>: [C <select-file>]

Two different existing disk drives must be specified in any case.

C <select-file>: Only the files on the first disk which are listed in <select-file> are compared to the corresponding files on the second disk. (Note that all files on the first disk are processed if the "C" switch is not entered.) <select-file> may be any valid ISIS-II disk file or input device name. The selection file may contain any number of file names without drive specifications in arbitrary order. The file names must be separated by one or more of the following characters:

* Spaces
* Carriage-return line feed pairs
* "Escape" characters
* TABs
* Any other control characters

If a file specified by the selection file is not found on the first disk, an appropriate error message is output; execution is continued for the next submitted

file name. No "wild card" file names ("*.SRC") are permitted.

REMARKS:

"CMPDSK" does not allow access to a file with the "F" attribute.

The names of the files to be processed may be entered directly on the console if "CMPDSK" is invoked with the switch "C:CI:". This is particularly advantageous if only a few files have to be handled. Note that the execution of "CMPDSK" is started only after the file names input on the console were actually entered (by pressing "Return" or "Escape"). Several file names may be specified within one input line, and the number of input lines is unlimited. The execution must be terminated by entering "CNTL-Z" after all files were processed.

A selection disk file may be generated from scratch with "CREDIT" or "CREATE". It is probably more convenient to generate a suitably formatted copy of the disk directory by a "DIRFIL" call and to edit this file with CREDIT if a large number of the files on one disk are to be processed. The same file can be used for all file selecting programs within this package ("ATTSET", "CMPDSK", and "CPYDSK"), no matter on which drive the disk to be processed is mounted. (This is why the selection file must not contain any drive references.)

A multiple specification of a file name within the selection file does not matter. If the "C" command is issued multiplely within the command line, all selection file definitions except the first one are ignored.

Appendix 6.3: File Comparison Utility COMP

This program compares two disk files on a byte-by-byte basis. The result is one of the following three messages output on the console:

FILES ARE IDENTICAL:
Each byte of one file is identical to the corresponding byte of the other file.

FILES ARE DIFFERENT:
One byte differing from the corresponding byte in the other file was discovered; execution was terminated. (Note that "COMP" does not continue checking the files

after it detected two different bytes. The message is, accordingly, the same no matter if only one byte is different or if totally different files were compared.)

FILES HAVE DIFFERENT SIZES:
"COMP" reads blocks of up to 16 Kbytes from each file. For files whose sizes exceed the buffer length of 16 KBytes, a difference in file size is only detected and reported when the last pair of buffers is read. "COMP" terminates its operation immediately in this case. Since "COMP" may have been preempted due to differing file contents before it could read the end of the two files to be compared, a "FILES ARE DIFFERENT" message may have been issued although a "DIFFERENT SIZES" message would have been more appropriate.

The messages referring to differences between the two files are preceded by the string "#####" which is very conspicuous on a CRT console. In addition, a "beep" is output.

PROGRAM CALL: COMP <disk file 1> TO <disk file 2>

Both file names must indicate disk files; the files may reside on the same or on different disks. Any attempt to compare a file to itself ("COMP MYFILE.EXT TO MYFILE.EXT") causes a fatal error and abortion of 'COMP". No "wild card" characters are permitted.

The following program calls are permitted:

COMP :F<n>:<file1> TO :F<m>:<file2>
Standard call; any disk drives and file names may be specified.

COMP <file1> TO <file2>
A default device name ":F0:" is assumed if no device is specified.

COMP :F<n>:<file1> TO :F<m>:
The file <file1> on disk <n> is compared to a file with the same name on disk <m>. An omitted device specification is interpreted as ":F0:".

COMP :F<n>:<file1>
The file <file1> on disk <n> is compared to the file <file1> on disk 0. Note: "COMP <file1>" is not permitted as the file would be compared to itself.

Appendix 6.4: Enhanced File Copy Utility COPYCP

This program combines a COPY action with the action of "COMP". The input file is copied to an output file, and both files are compared by means of the algorithms used in "COMP". Therefore, this program guarantees that the copied file is actually identical to the source file, and that there are no disk defects which would prevent the copy from being read. "COPYCP" reports its current status and the results of the comparison on the console. In the (most probable) case that there were no problems, its output reads:

"<file1> COPIED TO <file2> --> FILES ARE IDENTICAL"

The first part of this message is output after the input file was copied, the second part (following the arrow) after the comparison. This second part is replaced by one of the messages issued by "COMP" if some kind of read-write error happened.

Note: If a not write protected file with the name specified for the output file already exists on the specified output drive, it is deleted without further notice and replaced by a copy of the input file. A fatal error occurs, and the execution of "COPYCP" is aborted, if this file has its write protect attribute set. "COPYCP" does not transfer the attributes of the input file to the output file.

PROGRAM CALL: COPYCP <inputfile> TO <outputfile>

Both file names must be valid ISIS-II disk file names. No "wild card" characters may be used.

The following program calls are permitted:

COPYCP :F<n>:<file1> TO :F<m>:<file2>
Standard call; any disk drives and file names may be specified.

COPYCP <file1> TO <file2>
A default device name ":F0:" is assumed if no device is specified.

COPYCP :F<n>:<file1> TO :F<m>:
The file <file1> on disk <n> is copied to a file with the same name on disk <m>. An omitted device specification is interpreted as ":F0:".

COPYCP :F<n>:<file1>
The file <file1> on disk <n> is copied to the file <file1> on disk 0. Note: "COPYCP <file1>" is not permitted since the file would be copied to itself.

Appendix 6.5: Disk Copy Utility CPYDSK

This program copies either all files on the first disk specified in the program call to the second disk, or a number of files on this disk whose names are contained in a selection list. The latter feature constitutes its major advantage over the ISIS-II system program "COPY", aside from the fact that "CPYDSK" compares each copy to the pertinent source file. The algorithms and messages employed by CPYDSK are identical to those used in "COPYCP". In addition, the file attributes of the source files may either be transferred to the output disk, or be modified arbitrarily. All files copied and the results of the subsequent comparison are listed on the console. In case of a copy error, i.e., if an output file differs from its source file, execution is immediately aborted. A not write protected file on the output disk is replaced without further notice by a copy of a file on the input disk which has the same name; if the file on the output disk is write protected, however, an appropriate message is output on the console, and "CPYDSK" proceeds to the next file, not affecting the write protected file. Upon program termination, the total numbers of files copied and of write protected files encountered on the output disk are displayed, and, if applicable, a warning that files specified by the selection file have not been found on the input disk.

PROGRAM CALL: CPYDSK :F<m> TO :F<n>: [switches]

Two different valid disk drives must be specified in any case. The first drive (:F<m>:) contains the input disk, the second (:F<n>:), the output disk.

[switches] may be any arbitrary optional sequence of one or more of the following commands in upper- or lowercase, separated by at least one space:

A: All files on the input disk are to be copied to the output disk. They are re-arranged, however, in alphabetical order on the output disk.

C <select-file>: Only the files specified within <select-file> are processed. (The entire input disk is copied if the "C" switch is not used.) <select-file> may be any valid ISIS-II disk file or input device name. The selection file may contain any number of file names without drive specifications in arbitrary order. The order of the files specified by a selection file overrides their order on the source disk or an "A" switch. The file names in the selection file must be separated by one or more of the following characters:

* Spaces
* Carriage-return line feed pairs
* "Escape" characters
* TABs
* Any other control characters

If a file listed in the selection file is not found on the input disk, an appropriate error message is issued; no further action takes place, and the execution continues with the next submitted file name. Note that no output file is opened on the output disk in this case. The missing file will be added at the end of the disk directory rather than in the position specified by the selection file if it is later copied from another input disk. No "wild card" file names ("*.SRC") are permitted in the selection file.

N: The attributes of the files of the input disk are ignored; only attributes explicitly set by one or more of the following switches are set on the output files:

I{0|1}: This switch resets (0) or sets (1) the "invisible" attribute of the copied files. Either "0" or "1" must immediately follow the "I".

S{0|1}: This switch resets (0) or sets (1) the "system" attribute of the copied files. Either "0" or "1" must immediately follow the "S".

W{0|1}: This switch resets (0) or sets (1) the "write protect" attribute of the copied files. Either "0" or "1" must immediately follow the "W".

REMARKS:

Attributes which are explicitly set or reset with the "CPYDSK" call are valid for all files of the output disk, no matter what their state on the input disk was. The "N" switch is

equivalent to the sequence "W0 S0 I0". The attributes of the files generated on the output disk are otherwise set identically to those of the corresponding source files if none of the attribute switches was specified.

"CPYDSK" does not permit any modification of the "format" ("F") attribute, nor does it allow any access to a file with the "F" attribute set. This feature was provided in order to protect the ISIS-II system files from accidental deletion.

The names of the files to be processed may be entered directly on the console if "CPYDSK" is invoked with the switch "C:CI:". This is particularly advantageous if only a few files with totally different names have to be copied (and therefore there is no possibility of using the "wild card" feature of "COPY") but if the generation of a disk file would not pay. Note: The execution of "CPYDSK" is started only after the file names input on the console were actually entered (either by "Return" or by "Escape"). Several file names may be specified within one input line, and the number of input lines is unlimited. The execution must be terminated by entering "CNTL-Z" after all files were processed.

A selection disk file may be generated from scratch with "CREDIT" or "CREATE". It is probably more convenient to generate a suitably formatted copy of the disk directory by a "DIRFIL" call and to edit this file with CREDIT if a large number of the files on one disk are to be processed. The same file can be used for all file selecting programs within this package ("ATTSET", "CMPDSK", and "CPYDSK"), no matter on which drive the disk to be processed is mounted. (This is why the selection file must not contain any drive references.)

A multiple specification of a file name within the selection file does not matter. In the case of multiple but contradictive switch definitions within one program call, the last definition entered is considered valid. However, a multiple specification of the "C" switch is prohibited and will cause a fatal command error.

Appendix 6.6: File Generation Utility CREATE

The program "CREATE" permits the generation of new disk files. It was written in order to overcome the large time overhead inherent with the execution of "CREDIT", Intel's CRT text editor, which is particularly inconvenient if only short disk files are to be created. Any text entered on the console is copied to the output file specified with the "CREATE" call.

Utilizing the line editing features provided by ISIS-II or RXISIS-II, "CREATE" allows still corrections of the input line. The input line is added to the output file only when the "Return" or the "Escape" key is pressed. No subsequent editing is possible within "CREATE". "CREATE" terminates if "CNTL-Z" is entered on the console.

PROGRAM CALL: CREATE <filename>

REMARKS:

Any valid ISIS-II file or device name (not necessarily a disk file) may be used with "CREATE". An already existing disk file with the specified name is deleted without notice if it is not write protected. The execution of "CREATE" is aborted if an existing target file is write protected.

The echo output provided by RXISIS-II (and utilized by "CREATE") may differ from the echo output generated by "CREDIT". TAB characters, for example, are not properly displayed on the console CRT although they are correctly entered into the output file. (Note that control characters like a TAB must be preceded by CNTL-P under RXISIS-II in order to be accepted by the Terminal Handler.)

The line termination characters (carriage-return plus one line feed added by the system, or "Escape") are entered into the output file. The use of "Escape" as a line termination code should be avoided, though.

Warning: Entering "CNTL-Z" in order to terminate the "CREATE" session deletes the contents of the line editing input buffer. Be sure to provide a "Return" (or an "Escape") at the end of the last input line before entering "CNTL-Z"!

Appendix 6.7: Disk Directory List Utility DIRFIL

This program permits the generation of a list of the files contained in a disk directory. The file generated by "DIRFIL" may be used immediately as a selection file for "ATTSET", "CMPDSK", or "CPYDSK", or it may be edited before with CREDIT. Each file name is left-adjusted within a separate output line which is terminated by a carriage-return - line feed pair.

PROGRAM CALL: DIRFIL :F<n>: [TO <outputfile>] [A]

The command must indicate an existing disk drive; the directory of the disk mounted on this drive is processed by "DIRFIL". Note: Even drive 0 must be specified explicitly as ":F0:"!

<outputfile> may be any valid ISIS file or device name; the console output is assumed as a default if the output file specification is omitted.

"A" is an optional switch. "DIRFIL" lists the directory contents in their original order if the "A" switch is not set; this list is sorted alphabetically if "A" is specified.

REMARKS:

Files which have the "format" ("F") attribute set are ignored by "DIRFIL".

Appendix 6.8: File Conversion Utility HEXCHK

"HEXCHK" converts any arbitrary file into a legible and printable representation. It generates an output in three columns which may be optionally routed to a disk file; the default output device is the console. The first output column contains a (decimal) line number with up to five digits. Within the second column, eight bytes per line are converted into their hexadecimal representation (00 through FF for each byte). Within the third column, the same eight bytes are interpreted as ASCII-coded characters. If the code does not correspond to a printable character, the following conversions are performed:

CONTROL CHARACTERS (< 20H):	"^" + CHARACTER
RUBOUT (7FH):	RO
NON-ASCII CHARACTERS:	
(7FH < BYTE < A0H):	"$" + CHARACTER
(9FH < BYTE < FFH):	"#" + CHARACTER
(FFH):	FF

EXAMPLES:

BYTE:		OUTPUT:
00H		^@
0AH	(CNTL-J)	^J
1BH	(ESCAPE)	^[
61H		a
81H		$A
A1H		#A
E1H		#a
FFH		FF

The parallel output of hexadecimal and ASCII representation was provided in order to improve clearness, particularly for object program files which contain not only hexadecimal code but also ASCII-coded strings.

PROGRAM CALL: HEXCHK <inputfile> [TO <outputfile>]

Any valid ISIS file or device name may be specified for <inputfile> and <outputfile>. The output is routed to the console if <outputfile> is omitted.

REMARKS:

Any type of file may be processed with "HEXCHK". No restrictions apply to the code accepted by "HEXCHK"; each byte of the input file is processed no matter what its value is.

Appendix 6.9: File Listing Utility LIST

"LIST" is a program primarily intended for the generation of printer listings of ASCII-coded source files. Still, its output can be routed to any arbitrary ISIS-II file (including disk files and the console). Its main advantages compared to the "COPY" command are:

* Shorter calling sequence.
* TABs embedded in the input file are correctly processed.
* Page headers including the input file name and a page number and form feeds can be generated if required.

PROGRAM CALL: LIST <inputfile> [TO <outputfile>] [N]

<inputfile>: any correct ISIS-II file name (no "wild card" characters ("*.SRC") are permitted).

<outputfile>: any correct ISIS-II file name; must not be identical to <inputfile>. Default output file: :TO: (line printer).
N: optional switch; suppresses header and form feed generation.

REMARKS:

The program call may be entered as well in uppercase as in lowercase letters; the input file name in the page header is always output in uppercase letters.

If the switch "N" has not been specified, a form feed is issued prior to the output file generation. Therefore, the printer need not be adjusted to its top of form position prior to printing any file generated with "LIST".

Form feed characters (CNTL-L) embedded in the source file cause the output of a new page in either (header or no header) mode.

No restrictions apply to the lengths of the input lines; still, output lines are subdivided into two or more lines if their lengths exceed 122 characters.

Pages are numbered beginning with " 1"; page numbers up to "999" are possible. The thousands are omitted if more than 999 output pages are generated (therefore: "998", "999", "000", "001",...)

EXAMPLES:

LIST :F1:MYFILE.SRC
The file "MYFILE.SRC" on disk 1 is printed on the line printer with page header and form feed generation.

LIST FILE.SRC TO FILE.LST N
The file "FILE.SRC" on disk 0 is copied to a file "FILE.LST" on the same disk. All TABs in the source file are replaced by an appropriate number of spaces. Due to the switch "N", no headers are generated.

LIST :CI: N

This command permits the use of the printer in direct "typewriter" mode. Any text input from the console is printed as soon as the line is terminated by a "carriage-return" (or if its length exceeds the maximum console input line length of 122 characters). The input text is also echoed to the console CRT; the usual line editing features are maintained. Program execution must be terminated by entering "CNTL-Z" on the console keyboard. This command permits the addition of comments, etc., to the printer output without the need of first generating a disk file.

LIST MYFILE.SRC TO :CO: N

This command produces a program listing on the console device. For this purpose, however, the program "SHOW" or the RXISIS-II function "@" are better suited.

Appendix 6.10: File Display Utility SHOW

This program is provided for a quick check of ASCII files on the console CRT. It provides, in contrast to "COPY", correct TAB processing and marks additional lines which have been generated by the subdivision of too long input lines.

PROGRAM CALL: SHOW <filename>

<filename> can be any valid ISIS filename designating a file or device capable for input to the system. No "wild card" characters are permitted.

REMARKS:

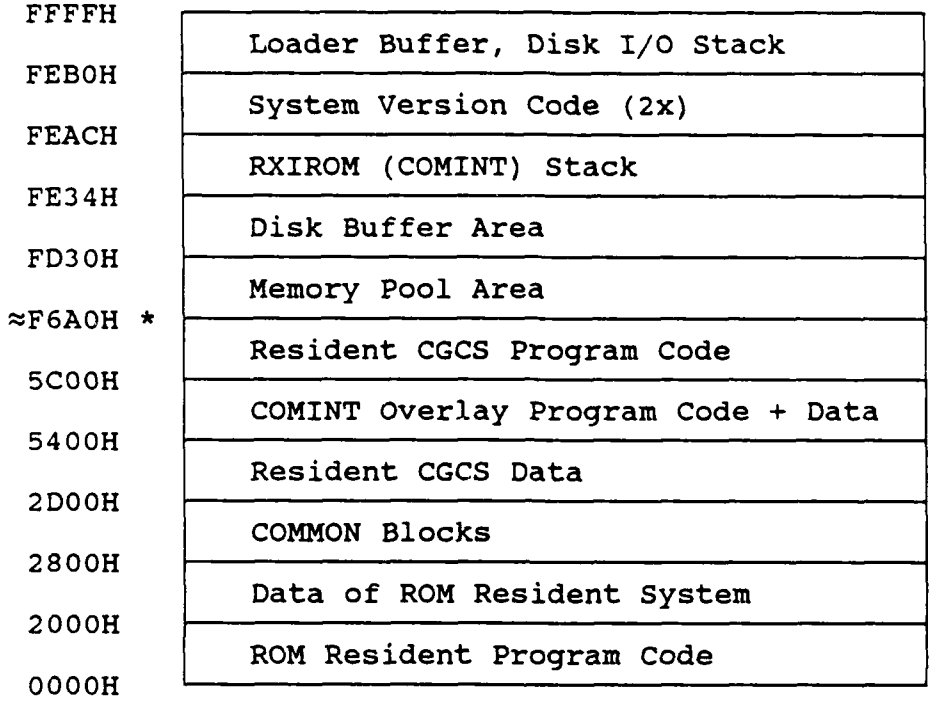
The output on the console CRT can be interrupted at any time by entering "CNTL-S" on the console keyboard. It is resumed without loss of information as soon as "CNTL-Q" is entered.

Input lines with excessive lengths are subdivided into two or more output lines which fit onto the CRT screen. Each additional output line generated in this way is marked by an arrow ("--->") at the left margin and is indented by 8 characters.

The functions of "SHOW" have been integrated into the RXISIS-II Command Line Interpreter. "SHOW" has therefore effectively been replaced by the RXISIS-II command "@". (A similar function is now also available with version 4.3 of ISIS-II.)

Appendix 7: CGCS Memory and I/O Maps

Appendix 7.1: Memory Map



* This boundary is most subject to changes due to program modifications. The value given applies to Version 2.4.

Appendix 7.2: I/O Map

```
20H ... A/D Converter Control/Status Register, low byte
21H ... A/D Converter Control/Status Register, high byte
24H ... A/D Converter Multiplexer Address Register
26H ... A/D Converter Output Data Register, low byte
27H ... A/D Converter Output Data Register, high byte

40H ... D/A Converter Channel 0, low byte
41H ... D/A Converter Channel 0, high byte
42H ... D/A Converter Channel 1, low byte
43H ... D/A Converter Channel 1, high byte
...
5EH ... D/A Converter Channel 15, low byte
5FH ... D/A Converter Channel 15, high byte
```

B0H ... I/O Expansion Board Base Address

B4H ... Motor Direction Relay Input
B5H ... Motor Direction Relay Output
B6H ... Controller Selection Relay Output

C0H - FFH ... CPU Board I/O Addresses

Various I/O ports on the iSBC 80-24 CPU and iSBC 517 I/O Expansion boards are used by system routines, e.g., by the Terminal Handler and the alternative FORTRAN floating-point routines.

Appendix 8: System Tasks

Appendix 8 lists all primary tasks within the system; it does not include tasks which are dynamically created at runtime by any primary task. SUSPEND information is given for the genuine CGCS tasks; it indicates whether a task may be suspended with the DEBUG Suspend command. It is generally prohibited to suspend any iRMX-80 System or Interface task!

Appendix 8.1: ROM Resident System Tasks

Task RXIROM: ROM resident root of RXISIS-II and the CGCS Command Interpreter.

Entry Point:	RXIROM
Stack Length:	50, extended to 120 by the CGCS
Priority:	250
Task Descriptor:	RXIRTD
Extra:	20

Task RQTHDI: Alternative Input Terminal Handler.

Entry Point:	RQTHDI
Stack Length:	40
Priority:	97
Task Descriptor:	THDITD
Extra:	0

Task RQTHDO: Alternative Output Terminal Handler.

Entry Point:	RQTHDO
Stack Length:	40
Priority:	113
Task Descriptor:	THDOTD
Extra:	0

Task RQLOAD: Alternative Loader Task.

Entry Point:	RQLOAD
Stack Length:	60
Priority:	140
Task Descriptor:	LOADTD
Extra:	0

Task DISKIO: iRMX-80 Disk I/O Task.

Entry Point:	RQPDSK
Stack Length:	48
Priority:	129
Task Descriptor:	RQDIOD
Extra:	0

Task : Unnamed Disk Controller Task.

Entry Point:	RQHD4
Stack Length:	80
Priority:	33
Task Descriptor:	CNTLTD
Extra:	0

Appendix 8.2: iRMX-80 System Tasks in the CGCS

Task RQFMGR: Free Space Manager.

Entry Point:	RQFMGR
Stack Length:	40
Priority:	50
Task Descriptor:	RQFSMD
Extra:	0

Task DIRSVC: Disk Directory Services.

Entry Point:	RQPDIR
Stack Length:	48
Priority:	200
Task Descriptor:	RQDRSD
Extra:	0

Appendix 8.3: FORTRAN - iRMX-80 Interface Tasks

Task FXCFLG: Flag Interrupt Generation Task.

Entry Point:	FXCFLG
Stack Length:	36
Priority:	149
Task Descriptor:	FXCFTD
Extra:	0

Task INDATX: Input Interface Task.

Entry Point:	FXINTI
Stack Length:	184
Priority:	134
Task Descriptor:	INDTTD
Extra:	18

Task OUTDTX: Output Interface Task.

Entry Point:	FXINTO
Stack Length:	200
Priority:	135
Task Descriptor:	OUTDTD
Extra:	18

Task FXDISK: Disk I/O Interface Task.

Entry Point:	FXDISK
Stack Length:	38
Priority:	133
Task Descriptor:	DISKTD
Extra:	0

Task FXTIME: System Timer Task.

Entry Point:	FXTIME
Stack Length:	34
Priority:	34
Task Descriptor:	TIMETD
Extra:	0

Appendix 8.4: Controller Tasks

Task CMMDEX: Command Executor Task.

Entry Point:	CMMDEX
Stack Length:	120
Priority:	240
Task Descriptor:	CMEXTD
Extra:	20
Suspend:	no

Task MEASDO: Measured Data Output Task.

Entry Point:	MEASDO
Stack Length:	120
Priority:	220
Task Descriptor:	MEASTD
Extra:	20
Suspend:	yes

Task CMFINP: Command File Input Task.

Entry Point:	CMFINP
Stack Length:	50
Priority:	230
Task Descriptor:	CMFITD
Extra:	20
Suspend:	yes

Task CMFOUT: Command File Output Task.

Entry Point:	CMFOUT
Stack Length:	50
Priority:	251
Task Descriptor:	CMFOTD
Extra:	20
Suspend:	yes; for short time only

Task DSKOUT: Data Disk File Output Task.

Entry Point:	DSKOUT
Stack Length:	50
Priority:	180
Task Descriptor:	DSKOTD
Extra:	20
Suspend:	yes

Task DIACNT: Diameter Controller Task.

Entry Point:	DIACNT
Stack Length:	120
Priority:	160
Task Descriptor:	DIACTD
Extra:	20
Suspend:	yes

Task ANACNT: Analog Data Controller Task.

Entry Point:	ANACNT
Stack Length:	60
Priority:	150
Task Descriptor:	ANACTD
Extra:	0
Suspend:	no

Appendix 9: Routine Names

Appendix 9.1: FORTRAN-iRMX-80 Interface Routine Names

iRMX-80 Control Routines - Library FRXMOD.LIB

NAME	TYPE	FUNCTION	CHAPTER
FXSEND	subr	non-reentrant msg. sending rout.	5.2.1.1
FXWAIT	subr	non-reentr. msg. receiving rout.	
FXACPT	subr	non-reentr. msg. receiving rout.	
FRSEND	subr	reentrant message sending rout.	5.2.1.2
FRWAIT	subr	reentr. message receiving rout.	
FRACPT	subr	reentr. message receiving rout.	
FRINIT	subr	initialization routine	
FRCRSP	func	check for response message	
FRCXCH	subr	exchange creation routine	5.2.1.3
FRDLVL	subr	interrupt level disabling rout.	
FRDTSK	subr	task deletion routine	
FRDXCH	subr	exchange deletion routine	
FRELVL	subr	interrupt level enabling routine	
FRRESM	subr	task execution resuming routine	
FRSUSP	subr	task execution suspending rout.	
FRACTV	func	task descriptor of running task	
FXCFLG	task	flag interrupt creation task	5.2.1.4
FXCRFE	subr	create flag interrupt exchange	
FXDLFE	subr	disable flag interrupt exchange	
FRACCS	subr	access common resources	5.2.1.5
FRRELS	subr	release common resources	
FRINAR	subr	create an access control exch.	
FXSYSE	subr	system error reporting routine	5.2.1.6
FRIFSM	subr	Free Space Manager initializ.	5.2.1.7

Console, Printer, and Buffer Input/Output Routines - Libraries FIORMX.LIB, FIOISS.LIB, FIORXI.LIB, and FIORXR.LIB

NAME	FUNCTION	CHAPTER
FRIOST	initialization routine for I/O funct.	5.2.2.1
FRDATI FRSTRI FRDTBI FRSTBI	data input routine (from console) character string input routine (cons.) data input routine (from user buffer) character string input routine (buffer)	5.2.2.2
FRDATO FRSTRO FRDTPR FRSTPR FRDTBO FRSTBO	data output routine (to console) char. string output routine (to cons.) data output routine (to printer) char. string output routine (to print.) data output routine (to user buffer) char. string output routine (to buffer)	5.2.2.3
FRINMD FROUTM FRPRMD FRINPR FRCLRO FRSPTO FRMCHG	input mode selection routine output mode selection routine (console) printer mode selection routine input prompt string modification CRT screen clearing routine printer timeout setting routine LOGICAL*1 function: output mode changed	5.2.2.4
FRCSTR	control string building routine	5.2.2.5
FRSTHX FRFXIN FXFLIN FRHXOT FRFXOT FXFLOT	conversion ASCII-INTEGER*1 conversion ASCII-INTEGER*2 conversion ASCII-REAL conversion to hexadecimal ASCII string conversion INTEGER-ASCII conversion REAL-ASCII	5.2.2.6

Disk Interface Routines - Libraries FXDISK.LIB and FXDSKI.LIB

NAME	TYPE	FUNCTION	CHAPTER
FROPEN	subr	disk file opening routine	5.2.3.1
FRREAD	subr	read data from disk file	5.2.3.2
FRWRTE	subr	write data to disk file	5.2.3.3
FRSEEK	subr	perform SEEK operation	5.2.3.4
FRCLSE	subr	disk file closing routine	5.2.3.5
FRLOAD	subr	load code from disk file	5.2.3.6
FRATTR FRDELT FRRNME	subr subr subr	disk file attribute setting disk file deleting routine disk file renaming routine	5.2.3.7
FREXIT	subr	exit to operating system	5.2.3.8
FRDSTA	func	check the status of a disk I/O operation	5.2.3.9
FXDSKE	subr	disk error message generation	5.2.3.10

General Utility Routines - Library FXUTIL.LIB

NAME	TYPE	FUNCTION	CHAPTER
FXTIME	task	timer task	5.2.4.1
FRSETT	subr	reset timer	
FXOCNS	subr	open console file	5.2.4.2
FXRCNS	subr	read from console file	
FXCCNS	subr	close console file	
FRCMPS	func	string comparison routine	5.2.4.3
FRCVUC	subr	string conversion to uppercase	
FRPOKE	subr	transfer of data to memory	5.2.4.4
FRPEEK	subr	transfer of data from memory	
FRADDR	func	returns address of parameter	
FRADD	subr	overflow-protected addition rout.	5.2.4.5
FRMULT	subr	overflow-protected multiplication	
FRSHFT	subr	scaling by powers of 2	
FRPIDC	subr	PID controller routine	5.3.2.1

Appendix 9.2: Controller Routine Names

The following table lists the names of all routines which do not belong to iRMX-80 or Interface libraries. The name of the source file which holds the routine is either equal to the routine's name, plus the extension ".SRC", or it is equally derived from the name given in parentheses. The main chapter in this documentation where references to a particular routine occur is specified, too. The following abbreviations were used:

A ... Assembly language module.
B ... (FORTRAN) BLOCKDATA program.
D ... Data module.
F ... FORTRAN module.
R ... Subroutine or FORTRAN FUNCTION.
T ... Task or main routine of a task.

ANACNT	T-F	Analog Data Controller Task (5.3.2.3.1)
ANAINI	R-A	Analog Data Input Initialization routine (ANAINP) (5.3.2.2)
ANAINP	R-A	Analog Data Input routine (5.3.2.2.2)
ANAOPT	R-A	Analog Data Output routine (5.3.2.2.4)
ANOMAL	R-F	Anomaly Compensation routine (5.3.2.2.2)
BEEP	R-F	Beeping Routine (AUXCOM) (5.3.1.2)
BITCNT	R-A	Bit Counting routine (5.3.1.3.17)
BLKDTA	B-F	CZOOVD Data Initialization BLOCKDATA program (5.3.1.3)
CALCUL	R-F	Calculator Utility routine (5.3.1.3.11)
CHKAN1	R-F	Operator Answer Checking routine (MENOUT) (5.3.1.3.4)
CHKANS	R-F	Operator Answer Checking routine (AUXCOM) (5.3.1.2)
CHKDTB	R-F	Check Diameter Table routine (DIACNT) (5.3.2.2.3)
CHKFNM	R-A	File Name Checking routine (5.3.1.3, 5.3.1.4.1)
CLEARO	R-F	Conditional Command Clearing overlay routine (5.3.1.3.21)
CLIPRL	R-F	Input Line Clearing Routine (AUXCOM) (5.3.1.2)
CLRBUF	R-F	Buffer Clearing routine
CLRSCR	R-F	Scrolled Screen Area Clearing routine (5.3.1.3.4)
CLSFIL	R-F	File Closing Routine (AUXCOM) (5.3.1.3)
CMFINP	T-F	Command File Input Task (5.3.1.6)
CMFOUT	T-F	Command File Output Task (5.3.1.7)
CMMDEX	T-F	Command Executor Task (5.3.1.4)
CNTRL	R-A	Control Mode Determining routine (5.3.1.4)
COMINT	T-F	Command Interpreter (5.3.1.3)
COMMEN	R-F	Comment Entry routine (5.3.1.3.3)
CONDIT	R-F	Conditional Command Entry routine (5.3.1.3.14)
CREATE	R-A	CGCS System Creation routine (CZINIT) (5.3.1.3)
CZINIT	R-A	CGCS Initialization Routine (5.3.1.3)

CZOV01	B-F	Overlay Identification BLOCKDATA module (SETPAR)
CZOV02	B-F	Overlay Identification BLOCKDATA module (SETVAR)
CZOV03	B-F	Overlay Identification BLOCKDATA module (COMMEN)
CZOV04	B-F	Overlay Identification BLOCKDATA module (MENOUT)
CZOV05	B-F	Overlay Identification BLOCKDATA module (OPMODE)
CZOV06	B-F	Overlay Identification BLOCKDATA module (DEBUG0)
CZOV07	B-F	Overlay Identification BLOCKDATA module (DEBUG1)
CZOV08	B-F	Overlay Identification BLOCKDATA module (FRAME)
CZOV09	B-F	Overlay Identification BLOCKDATA module (FILES)
CZOV10	B-F	Overlay Identification BLOCKDATA module (REQCMF)
CZOV11	B-F	Overlay Identification BLOCKDATA module (CALCUL)
CZOV12	B-F	Overlay Identification BLOCKDATA module (DATAFI)
CZOV13	B-F	Overlay Identification BLOCKDATA module (EXICZO)
CZOV14	B-F	Overlay Identification BLOCKDATA module (CONDIT)
CZOV15	B-F	Overlay Identification BLOCKDATA module (DISPLY)
CZOV16	B-F	Overlay Identification BLOCKDATA module (DOCUMT)
CZOV17	B-F	Overlay Identification BLOCKDATA module (DIRECT)
CZOV18	B-F	Overlay Identification BLOCKDATA module (RESOVL)
CZOV19	B-F	Overlay Identification BLOCKDATA module (INIDAT)
CZOV20	B-F	Overlay Identification BLOCKDATA module (PLOTOV)
CZOV21	B-F	Overlay Identification BLOCKDATA module (CLEARO)
CZOVER	D-A	System Version code
DASHES	R-F	Half Line Of Dashes Generating routine (FRAME) (5.3.1.3.8)
DASHLN	R-F	Full Line Of Dashes Generating routine (FRAME) (5.3.1.3.8)
DATAFI	R-F	Data File Maintenance routine (5.3.1.3.12)
DATIN	R-A	Special Input Interface routine (DATOUT) (5.2.2.9)
DATOUT	R-A	Special Output Interface routine (5.2.2.9)
DEBUG0	R-F	DEBUG routines, part 1 (5.3.1.3.6)
DEBUG1	R-F	DEBUG routines, part 2 (5.3.1.3.7)
DIACNT	T-F	Diameter Controller Task (5.3.2.2.1)
DIALIM	R-F	Diameter Square Limiting Routine (DIACNT) (5.3.2.2.1)
DIRECT	R-F	Disk Directory Display routine (5.3.1.3.17)
DISINT	R-A	Interrupt Disabling Routine (AUXASM) (5.3.1.4.7)
DISPLY	R-F	Variable Display routine (5.3.1.3.15)
DOCUMT	R-F	Documentation File Maintenance routine (5.3.1.3.16)
DSKOUT	T-F	Data File Output Task (DSKDAT) (5.3.1.8)
DUMP	R-F	Data Dump Triggering routine (DUMPDT) (5.3.1.4.6)
DUMPDT	R-F	Data Dump Generation routine (5.3.1.4.6)
ENINT	R-A	Interrupt Enabling Routine (AUXASM) (5.3.1.4.7)
ERRMSG	R-A	Error Message Output routine (AUXASM) (5.3.1.2)
EXICZO	R-F	CGCS Exit routine (5.3.1.3.13)
FILES	R-F	Output File Status Display routine (5.3.1.3.9)
FINDAD	R-A	Variable Address Finding routine (5.3.1.3.2)
FIRSTM	R-A	First module in Code area - used by MEMCHK (5.3.1.4.8)

FRAME	R-F	Console Output Mask Generation routine (5.3.1.3.8)
FRPIDC	R-A	Generic PID Controller routine (5.3.2.1)
FXUSIN	R-F	Initialization routine (INIT) (5.3.1.3)
IMULT	R-A	INTEGER multiplication routine (5.3.2.3)
INIDAT	R-F	Initial Data Input routine (5.3.1.3.16)
INIDTA	B-F	Built-In Data Initialization BLOCKDATA program (5.3.1.3)
INIPRT	R-A	Special Output Interface routine (DATOUT) (5.2.2.9)
LOVLAY	R-F	Overlay Loading routine (AUXCOM) (5.3.1.3)
LOWPAS	R-A	Low-Pass Filtering routine (5.3.2.2.5)
LSTRAM	D-A	Dummy routine: Last program code module
MAKEFN	R-A	File Name Building routine (AUXASM) (5.3.1.3)
MEASDO	T-F	Measured Data Output Task (5.3.1.5)
MEMCHK	R-A	Code Memory Checking routine (LSTRAM) (5.3.1.4.8)
MENOUT	R-F	Help Menu Output routine (5.3.1.3.4)
MESSGE	R-A	Message Output routine (AUXASM) (5.3.1.2)
MOTDIR	R-A	Motor Direction Output routine (5.3.2.2.3)
OPMODE	R-F	Operation Mode Entry routine (5.3.1.3.5)
OPNFIL	R-F	File Opening Routine (AUXCOM) (5.3.1.3)
PEEKDW	R-A	Data Retrieval routine (AUXASM) (5.3.1.3.1)
PLOTOV	R-F	Data Plotting Setup Overlay (5.3.1.3.20)
PLOTPR	R-F	Plot Data Collecting Routine (5.3.1.4.7)
PRETTA	R-A	Auxiliary Command Interpreter routine (AUXASM) (5.3.1.2)
PROMPT	R-A	Command Prompt Generation routine (AUXASM) (5.3.1.2)
QUITCM	R-F	Macro Command Quitting routine (5.3.1.3)
REACTV	R-A	Diameter Evaluation Reactivating routine (SHAPE) (5.3.2.2.5)
REQCMF	R-F	Command Output File Maintenance routine (5.3.1.3.10)
RESET	R-A	Diameter Controller Resetting routine (SHAPE) (5.3.2.2.4)
RESOVL	R-F	RESET Command Processing routine (5.3.1.3.18)
SETPAR	R-F	Parameter Setpoint Entry routine (5.3.1.3.1)
SETVAR	R-F	Variable Setpoint Entry routine (5.3.1.3.2)
SHAPE	R-A	Diameter Controller routine (5.3.2.2.3)
SHIFTB	R-F	Buffer Left Shifting routine (DEBUG0) (5.3.1.3.6)
SHIFTB	R-F	Buffer Left Shifting routine (PLOTOV) (5.3.1.3.20)
SPLITM	R-A	Mode Code Splitting routine (AUXASM) (5.3.1.4.1)
STARTP	R-A	Special Output Interface routine (DATOUT) (5.2.2.9)
STODAT	R-A	Data Storage routine (AUXASM) (5.3.1.4.1)
STRIN	R-A	Special Input Interface routine (DATOUT) (5.2.2.9)
STRIPN	R-F	Strip Binary Zeros routine (DIRECT) (5.3.1.3.17)

STROUT	R-A	Special Output Interface routine (DATOUT) (5.2.2.9)
TESTHD	R-A	Hardware Testing routine (CZINIT) (5.3.1.3)
TIMLIN	R-F	Top Of Screen Line Output routine (5.3.1.3, 5.3.1.3.8)
TRVMOD	D-A	Trivial Module; needed for system configuration
VARNM1	B-F	Auxiliary DEBUG COMMON Block Initialization (DEBUG0) (5.3.1.3.6)
WTOUTP	R-F	MEASDO Delaying routine (MEASDO) (5.3.1.5)
XCHDSK	R-F	Disk Exchange routine (AUXCOM) (5.3.1.3)

Appendix 10: COMMON Blocks

The following table shows the Fortran COMMON blocks used in the CGCS, arranged in increasing address order. For each block, its size and the names of the routines referencing it are specified.

LOCATED IN THE MAIN COMMON AREA:

/ANAOUT/	(32)	CMMDEX, PLOTPR, MEASDO, DSKOUT, ANACNT, INIDTA
/ANIPAR/	(52)	ANACNT, INIDTA, BLKDTA
/ANOMLY/	(8)	ANOMAL, BLKDTA
/ANOPAR/	(17)	ANACNT, INIDTA, BLKDTA
/AUXILD/	(62)	PLOTPR, INIDTA, BLKDTA
/CNDCNT/	(1)	CMMDEX, MEASDO, INIDTA
/COMMEX/	(10)	FXUSIN, COMINT, SETPAR, SETVAR, OPMODE, DEBUG0, DEBUG1, EXICZO, CONDIT, RESOVL, PLOTOV, CLEARO, CMMDEX, CMFINP, DIACNT
/COMMFL/	(10)	FXUSIN, COMINT, SETPAR, SETVAR, OPMODE, DEBUG0, DEBUG1, EXICZO, CONDIT, RESOVL, PLOTOV, CLEARO, CMFINP, CMFOUT
/CONLIM/	(2)	MEASDO, INIDTA, BLKDTA
/CRUC0P/	(12)	DIACNT, INIDTA, BLKDTA
/CRUC1P/	(12)	DIACNT, INIDTA, BLKDTA
/DEBUG/	(43)	FXUSIN, CMMDEX, MEASDO, DSKOUT, INIDTA
/DEBUGE/	(1)	COMINT, CMMDEX, CMFINP, INIDTA
/DIA10P/	(12)	DIACNT, INIDTA, BLKDTA
/DIA11P/	(12)	DIACNT, INIDTA, BLKDTA
/DIA20P/	(12)	DIACNT, INIDTA, BLKDTA
/DIA21P/	(12)	DIACNT, INIDTA, BLKDTA
/DIA30P/	(12)	DIACNT, INIDTA, BLKDTA

/DIA31P/	(12)	DIACNT, INIDTA, BLKDTA
/DISKFN/	(56)	OPNFIL, TIMLIN, FILES, REQCMF, DATAFI, DOCUMT, CMMDEX, INIDTA
/DOUTEX/	(10)	FXUSIN, DSKOUT
/ENDBGO/	(1)	MENOUT, OPMODE, FRAME, DIRECT, MEASDO, INIDTA
/INTRVL/	(2)	EXICZO, WTOUTP, INIDTA, BLKDTA
/MODE/	(1)	COMMEN, OPMODE, EXICZO, RESOVL, CMMDEX, MEASDO, DSKOUT, DIACNT, ANACNT, INIDTA
/OVLNM1/	(6)	LOVLAY, CZOVxx, INIDTA, BLKDTA
/OVRLAY/	(1)	COMINT, CZOVxx, FILES, INIDTA
/PLOTAD/	(16)	CMMDEX, PLOTPR
/REALDT/	(88)	CMMDEX, DUMPDT, PLOTPR, DIACNT, INIDTA
/RECORD/	(3)	COMINT, FILES, REQCMF, CMFOUT, INIDTA
/RESTDO/	(3)	FXUSIN, FRAME, EXICZO, CMMDEX, MEASDO, INIDTA
/RMPPAR/	(401)	EXICZO, CMMDEX, MEASDO, INIDTA
/SECFLG/	(1)	CMMDEX, ANACNT
/SETPT0/	(33)	FXUSIN, SETPAR, EXICZO, INIDAT, CMMDEX, PLOTPR, MEASDO, DSKOUT, DIACNT, ANACNT, INIDTA
/SETPT1/	(33)	FXUSIN, INIDAT, CMMDEX, MEASDO, DSKOUT, DIACNT, INIDTA
/TEMP1P/	(12)	ANACNT, INIDTA, BLKDTA
/TEMP2P/	(12)	ANACNT, INIDTA, BLKDTA
/TEMP3P/	(12)	ANACNT, INIDTA, BLKDTA
/TEST/	(1)	FXUSIN, ANACNT
/WAITEX/	(10)	FXUSIN, QUITCM, EXICZO, INIDAT, WTOUTP
/XTDCNT/	(48)	ANACNT, BLKDTA
/XTDDAT/	(2)	EXICZO, DOCUMT, DUMPDT, DUMP, INIDTA

/XTLSHP/ (4) CHKDTB, BLKDTA

/XTRADT/ (8) ANACNT, INIDTA, BLKDTA

TIED TO THE DATA AREA (USED BY ASSEMBLY LANGUAGE MODULES):

MODULE FXTIME:

/F0TIME/ (65) FXUSIN, COMINT, QUITCM, COMMEN, DATAFI, EXICZO, CMMDEX, DUMPDT, DUMP, CMFINP, CMFOUT, DSKOUT, DIACNT, ANACNT

MODULE DATOUT:

/IOFLAG/ (4) FXUSIN, COMINT, CLSFIL, OPNFIL, QUITCM, COMMEN, TIMLIN, FILES, REQCMF, DATAFI, EXICZO, DOCUMT, CMMDEX, DUMPDT, CMFINP, CMFOUT, DSKOUT, INIDTA

/DISKLC/ (4) CLSFIL, OPNFIL, FILES, REQCMF, DATAFI, DOCUMT, DIRECT, INIDTA

/DATE/ (8) FXUSIN, CLIPRL, TIMLIN, DATAFI

/RUNID/ (20) FXUSIN, TIMLIN, DATAFI

MODULE SHAPE:

/ANADAT/ (65) FXUSIN, RESOVL, CMMDEX, PLOTPR, MEASDO, DSKOUT, DIACNT, ANACNT, INIDTA

/DIAMET/ (2) CMMDEX, PLOTPR, MEASDO, DSKOUT, DIACNT, INIDTA

/LENGTH/ (2) COMMEN, RESOVL, CMMDEX, MEASDO, DSKOUT, INIDTA

/SCALE/ (72) SETPAR, INIDAT, RESOVL, CMMDEX, PLOTPR, MEASDO, DIACNT, INIDTA, BLKDTA

/AUXDIA/ (26) INIDAT, PLOTPR, DIACNT, BLKDTA

/ZEROWT/ (2) ANACNT

/GROWTH/ (4) PLOTPR

/DIATAB/ (256) CHKDTB

LOCATED ON TOP OF THE COMMAND INTERPRETER OVERLAY AREA:

/DBGCOM/ (21) DEBUG0, VARNM1, DEBUG1

/SCONDT/ (8) FXUSIN, BLKDTA

LOCATED CLOSE TO THE HIGH ADDRESS END OF THE RAM AREA (IN CONTROLLER ADDRESSABLE MEMORY)

/DSKBUF/ (128) DSKOUT, INIDTA

Appendix 11: Variable Names

Appendix 11.1: Most Important Variables

Name	Type	Size	Meaning
			Raw Analog Input Data (2 Byte Int.)
ITEMP1 *	I2	1	Heater #1 Temperature
ITEMP2 *	I2	1	Heater #2 Temperature
ITEMP3 *	I2	1	Heater #3 Temperature
ISEEDL *	I2	1	Seed Lift
ICRUCL *	I2	1	Crucible Lift
ISEEDR *	I2	1	Seed Rotation
ICRUCR *	I2	1	Crucible Rotation
IPOUT1 *	I2	1	Power Output #1
IPOUT2 *	I2	1	Power Output #2
IPOUT3 *	I2	1	Power Output #3
IWEIGH *	I2	1	Weight
IDWGHT *	I2	1	Diff. Weight
ISEEDP *	I2	1	Seed Position
ICRUCP *	I2	1	Crucible Position
IBASET *	I2	1	Base Temperature
IGASPR *	I2	1	Gas Pressure
CONTAC *	I2	1	Contact Device
ANALOG *	I2	8	Spare Analog Channels
			Measured Analog Data (2 Byte Int.)
MTEMP1 +	I2	1	Heater #1 Temperature
MTEMP2 +	I2	1	Heater #2 Temperature
MTEMP3 +	I2	1	Heater #3 Temperature
MSEEDL +	I2	1	Seed Lift
MCRUCL +	I2	1	Crucible Lift
MSEEDR +	I2	1	Seed Rotation
MCRUCR +	I2	1	Crucible Rotation
MPOUT1 +	I2	1	Power Output #1
MPOUT2 +	I2	1	Power Output #2
MPOUT3 +	I2	1	Power Output #3
MWEIGH +	I2	1	Weight
MDWGHT +	I2	1	Diff. Weight
MSEEDP +	I2	1	Seed Position
MCRUCP +	I2	1	Crucible Position
MBASET +	I2	1	Base Temperature
MGASPR +	I2	1	Gas Pressure
MCONTC +	I2	1	Contact Device
MANALG +	I2	8	Spare Analog Channels

Raw Analog Output Data (2Byte Int.)

PWR1IN	*	I2	1	Input Power (to SCR Controller) #1
PWR2IN	*	I2	1	Input Power (to SCR Controller) #2
PWR3IN	*	I2	1	Input Power (to SCR Controller) #3
SEEDLO	*	I2	1	Seed Lift
CRUCLO	*	I2	1	Crucible Lift
SEEDRO	*	I2	1	Seed Rotation
CRUCRO	*	I2	1	Crucible Rotation

Processed Analog Data (REAL)

DIAMET	*	R	1	Crystal Diameter
TEMP1	*	R	1	Heater #1 Temperature
TEMP2	*	R	1	Heater #2 Temperature
TEMP3	*	R	1	Heater #3 Temperature
SEEDL	*	R	1	Seed Lift
CRUCL	*	R	1	Crucible Lift
SEEDR	*	R	1	Seed Rotation
CRUCR	*	R	1	Crucible Rotation
POWER1	*	R	1	Power Output #1
POWER2	*	R	1	Power Output #2
POWER3	*	R	1	Power Output #3
WEIGHT	*	R	1	Weight
DWGHT	*	R	1	Diff. Weight
SEEDP	*	R	1	Seed Position
CRUCP	*	R	1	Crucible Position
BASTMP	*	R	1	Base Temperature
GASPR	*	R	1	Gas Pressure
PWRIN1	*	R	1	Power Input (to SCR Controller) #1
PWRIN2	*	R	1	Power Input (to SCR Controller) #2
PWRIN3	*	R	1	Power Input (to SCR Controller) #3
LENGTH	*	R	1	Crystal Length Grown
ADJDW	*	R	1	Anomaly Adjusted Diff. Weight

Current Setpoints (2 Byte Int.)

STDIAM	*	I2	1	Diameter
STTMP1	*	I2	1	Heater #1 Temperature
STTMP2	*	I2	1	Heater #2 Temperature
STTMP3	*	I2	1	Heater #3 Temperature
SETSL	*	I2	1	Seed Lift
SETCL	*	I2	1	Crucible Lift
SETSR	*	I2	1	Seed Rotation
SETCR	*	I2	1	Crucible Rotation
STPWRL	*	I2	1	Power Limit

PID Controller Parameters:

Seed Lift Motor

SLGAIN	I1	1	Gain
SLCNTL	I1	1	Control
SLPROP	I2	1	Proportional Multiplier
SLINT	I2	1	Integral Multiplier
SLDIFF	I2	1	Differential Multiplier
SLLIM	I2	1	Limit
SLTHET	I2	1	THETA value

Crucible Lift Motor

CLGAIN	I1	1	Gain
CLCNTL	I1	1	Control
CLPROP	I2	1	Proportional Multiplier
CLINT	I2	1	Integral Multiplier
CLDIFF	I2	1	Differential Multiplier
CLLIM	I2	1	Limit
CLTHET	I2	1	THETA value

Seed Rotation Motor

SRGAIN	I1	1	Gain
SRCNTL	I1	1	Control
SRPROP	I2	1	Proportional Multiplier
SRINT	I2	1	Integral Multiplier
SRDIFF	I2	1	Differential Multiplier
SRLIM	I2	1	Limit
SRTHET	I2	1	THETA value

Crucible Rotation Motor

CRGAIN	I1	1	Gain
CRCNTL	I1	1	Control
CRPROP	I2	1	Proportional Multiplier
CRINT	I2	1	Integral Multiplier
CRDIFF	I2	1	Differential Multiplier
CRLIM	I2	1	Limit
CRTHET	I2	1	THETA value

Temperature #1
(#2 and #3 analogously)

T1GAIN	I1	1	Gain
T1CNTL	I1	1	Control
T1PROP	I2	1	Proportional Multiplier
T1INT	I2	1	Integral Multiplier
T1DIFF	I2	1	Differential Multiplier

T1LIM	I2	1	Limit Value
			Diameter #1 (controls Temp. #1) (#3 analogously)
			Main Controller
GAIN10	I1	1	Gain
CNTL10	I1	1	Control
PROP10	I2	1	Proportional Multiplier
INT10	I2	1	Integral Multiplier
DIFF10	I2	1	Differential Multiplier
LIM10	I2	1	Limit Value
			Auxiliary Controller
GAIN11	I1	1	Gain
CNTL11	I1	1	Control
PROP11	I2	1	Proportional Multiplier
INT11	I2	1	Integral Multiplier
DIFF11	I2	1	Differential Multiplier
LIM11	I2	1	Limit Value
			Diameter #2 (controls Temp. #2) (#3 analogously)
			Main Controller
GAIN20	I1	1	Gain
CNTL20	I1	1	Control
PROP20	I2	1	Proportional Multiplier
INT20	I2	1	Integral Multiplier
DIFF20	I2	1	Differential Multiplier
LIM20	I2	1	Limit Value
			Auxiliary Controller
GAIN21	I1	1	Gain
CNTL21	I1	1	Control
PROP21	I2	1	Proportional Multiplier
INT21	I2	1	Integral Multiplier
DIFF21	I2	1	Differential Multiplier
LIM21	I2	1	Limit Value
			Crucible Lift
			Main Controller
C0GAIN	I1	1	Gain
C0CNTL	I1	1	Control

C0PROP	I2	1	Proportional Multiplier
C0INT	I2	1	Integral Multiplier
C0DIFF	I2	1	Differential Multiplier
C0LIM	I2	1	Limit Value
			Auxiliary Controller
C1GAIN	I1	1	Gain
C1CNTL	I1	1	Control
C1PROP	I2	1	Proportional Multiplier
C1INT	I2	1	Integral Multiplier
C1DIFF	I2	1	Differential Multiplier
C1LIM	I2	1	Limit Value
			Low-Pass Filter Values (0 ... 4)
ANIPAR(4)	I1		Heater #1 Temperature
ANIPAR(6)	I1		Heater #2 Temperature
ANIPAR(8)	I1		Heater #3 Temperature
ANIPAR(10)	I1		Seed Lift
ANIPAR(12)	I1		Crucible Lift
ANIPAR(14)	I1		Seed Rotation
ANIPAR(16)	I1		Crucible Rotation
ANIPAR(18)	I1		Power Output #1
ANIPAR(20)	I1		Power Output #2
ANIPAR(22)	I1		Power Output #3
ANIPAR(24)	I1		Weight
ANIPAR(26)	I1		Diff. Weight
ANIPAR(28)	I1		Seed Position
ANIPAR(30)	I1		Crucible Position
ANIPAR(32)	I1		Base Temperature
ANIPAR(34)	I1		Gas Pressure
ANIPAR(36)	I1		Contact Device
ANIPAR(38)	I1		Spare Channel #1
ANIPAR(40)	I1		Spare Channel #2
ANIPAR(42)	I1		Spare Channel #3
ANIPAR(44)	I1		Spare Channel #4
ANIPAR(46)	I1		Spare Channel #5
ANIPAR(48)	I1		Spare Channel #6
ANIPAR(50)	I1		Spare Channel #7
ANIPAR(2)	I1		Spare Channel #8
			Other System Control Parameters
ANOMLY	R	2	Anomaly Compensation Factors

Shape Controller

ALPHA		R	1	Diameter Evaluation Mode Parameter
XTLSHP		R	1	Crystal Shape Smoothing Parameter
CDIASQ	*	R	1	Square of Crucible Diameter
SDIASQ	*	R	1	Square of Seed Diameter
OXWGHT	*	R	1	Oxide Weight
RHOXTL	*	R	1	Crystal Spec. Weight (scaled)
RHOMLT	*	R	1	Melt Spec. Weight (scaled)
RHOOXI	*	R	1	Oxide Melt Spec. Weight (scaled)
SCRUCP	*	I2	1	Setpoint for Crucible Position
HEIGHT	*	R	1	Boric Oxide Melt Height in Crucible
GROWTH	*	R	1	Actual Growth Rate

Chart Recorder Output

EXTMP1	*	I2	1	Expanded Temperature 1
EXTMP2	*	I2	1	Expanded Temperature 2
EXTMP3	*	I2	1	Expanded Temperature 3
EXTMPB	*	I2	1	Expanded Base Temperature
OFFST1		R	1	Offset for Temperature 1 Expansion
OFFST2		R	1	Offset for Temperature 2 Expansion
OFFST3		R	1	Offset for Temperature 3 Expansion
OFFSTB		R	1	Offset for Base Temperature Exp.
RANGT1		R	1	Range for Temperature 1 Expansion
RANGT2		R	1	Range for Temperature 2 Expansion
RANGT3		R	1	Range for Temperature 3 Expansion
RANGTB		R	1	Range for Base Temperature Exp.
GRRATE	*	I2	1	Expanded Growth Rate
DIAERR	*	I2	1	Expanded Diameter Error
CRPERR	*	I2	1	Expanded Crucible Position Error
ZERO		I2	1	Location Holding Zero

Miscellaneous System Parameters

TEST		I1	1	Test Mode Flag
INTRVL		I1	1	Wait Interval for Data Display (>0)
DUMPIN		I1	1	Interval between Data Dumps
DUMPFL		I1	1	Data Dump Request Flag
DIASTA		I1	1	Diameter Evaluation Routine Status
CONLIM		I1	1	Limit Value for Contact Device
TIME	*	I2	1	System Time (Seconds Counter)
RAMPNG	*	I1	1	Number of Parameters Ramped
CNDCNT	*	I1	1	Number of Conditional Commands
DUMMY		I2	8	Scratchpad Locations

* Read-only parameter, do not change!
+ Parameters can only be changed in Test mode.

Appendix 11.2: Complete List of Variables, Sorted by Address

```
ZERO     I2      LOCATION HOLDING ZERO
DISKIO   T       TD FOR TASK DISKIO
RQTHDI   T       TD FOR TASK RQTHDI
RQTHDO   T       TD FOR TASK RQTHDO
RQLOAD   T       TD FOR TASK RQLOAD
COMINT   T       TD FOR TASK COMINT = RXIROM
RXIROM   T
PWR1IN   I2      INTEGER: POWER INPUT (TO SCR MODULE)
PWR2IN   I2
PWR3IN   I2
SEEDLO   I2      INTEGER: SEED LIFT OUTPUT
CRUCLO   I2      INTEGER: CRUCIBLE LIFT OUTPUT
SEEDRO   I2      INTEGER: SEED ROTATION OUTPUT
CRUCRO   I2      INTEGER: CRUCIBLE ROTATION OUTPUT
PLOTDT   I2   8  INTEGER: CHART RECORDER OUTPUT DATA
ANAOUT   I2  16  ARRAY OF INTEGER OUTPUT DATA
ANIPAR   I1  52  PARAMETER ARRAY FOR ANALOG INPUT ROUTINE
ANOMLY   R    2  ANOMALY CORRECTION PARAMETERS (TWO REAL)
ANOPAR   I1  17  PARAMETER ARRAY FOR ANALOG OUTPUT ROUTINE
OFFST1   R       TEMPERATURE OFFSET - HEATER TEMPERATURE I
OFFST2   R                 HEATER TEMPERATURE II
OFFST3   R                 HEATER TEMPERATURE III
OFFSTB   R                 BASE TEMPERATURE
RANGT1   R       TEMPERATURE CHART RECORDER OUTPUT RANGE - T I
RANGT2   R       HEATER TEMPERATURE II
RANGT3   R       HEATER TEMPERATURE III
RANGTB   R       BASE TEMPERATURE
EXTMP1   I2      EXPANDED HEATER TEMPERATURE I
EXTMP2   I2      EXPANDED HEATER TEMPERATURE II
EXTMP3   I2      EXPANDED HEATER TEMPERATURE III
EXTMPB   I2      EXPANDED BASE TEMPERATURE
DIAERR   I2      EXPANDED DIAMETER ERROR
CRPERR   I2      EXPANDED CRUCIBLE POSITION ERROR
GRRATE   I2      EXPANDED GROWTH RATE
DUMMY    I2   8  EIGHT DUMMY LOCATIONS
CNDCNT   I1      COUNTER FOR CONDITIONAL COMMANDS
CONLIM   I2      INTEGER: LIMIT VALUE FOR CONTACT DEVICE
C0GAIN   I1      CRUCIBLE LIFT CONTROLLER ARRAY: GAIN
C0CNTL   I1          CONTROL BYTE
C0PROP   I2          PROP. MULTIPL.
C0INT    I2          INT. MULTIP.
C0DIFF   I2          DIFF. MULTIP.
C0LIM    I2          LIMIT
CRUC0P   I2   6  CRUCIBLE LIFT CONTROLLER ARRAY
C1GAIN   I1      AUXILIARY CRUC. LIFT CONTROLLER: GAIN
C1CNTL   I1          CONTROL BYTE
C1PROP   I2          PROP. MULTIPL.
C1INT    I2          INT. MULTIPL.
```

```
C1DIFF   I2          DIFF. MULTIPL.
C1LIM    I2          LIMIT
CRUC1P   I2   6  AUXILIARY CRUCIBLE LIFT CONTROLLER
GAIN10   I1      MAIN DIAM. CNTL. I:   GAIN
CNTL10   I1          CONTROL BYTE
PROP10   I2          PROP. MULTIPL.
INT10    I2          INT. MULTIPL.
DIFF10   I2          DIFF. MULTIPL.
LIM10    I2          LIMIT
DIA10P   I2   6  MAIN DIAMETER CONTROLLER I
GAIN11   I1      AUX. DIAM. CNTL. I:   GAIN
CNTL11   I1          CONTROL BYTE
PROP11   I2          PROP. MULTIPL.
INT11    I2          INT. MULTIPL.
DIFF11   I2          DIFF. MULTIPL.
LIM11    I2          LIMIT
DIA11P   I2   6  AUXILIARY DIAMETER CONTROLLER I
GAIN20   I1      MAIN DIAM. CNTL. II:  GAIN
CNTL20   I1          CONTROL BYTE
PROP20   I2          PROP. MULTIPL.
INT20    I2          INT. MULTIPL.
DIFF20   I2          DIFF. MULTIPL.
LIM20    I2          LIMIT
DIA20P   I2   6  MAIN DIAMETER CONTROLLER II
GAIN21   I1      AUX. DIAM. CNTL. II:  GAIN
CNTL21   I1          CONTROL BYTE
PROP21   I2          PROP. MULTIPL.
INT21    I2          INT. MULTIPL.
DIFF21   I2          DIFF. MULTIPL.
LIM21    I2          LIMIT
DIA21P   I2   6  AUXILIARY DIAMETER CONTROLLER II
GAIN30   I1      MAIN DIAM. CNTL. III: GAIN
CNTL30   I1          CONTROL BYTE
PROP30   I2          PROP. MULTIPL.
INT30    I2          INT. MULTIPL.
DIFF30   I2          DIFF. MULTIPL.
LIM30    I2          LIMIT
DIA30P   I2   6  MAIN DIAMETER CONTROLLER III
GAIN31   I1      AUX. DIAM. CNTL. III: GAIN
CNTL31   I1          CONTROL BYTE
PROP31   I2          PROP. MULTIPL.
INT31    I2          INT. MULTIPL.
DIFF31   I2          DIFF. MULTIPL.
LIM31    I2          LIMIT
DIA31P   I2   6  AUXILIARY DIAMETER CONTROLLER III
INTRVL   I2      INTERVAL FOR MEASURED DATA OUTPUT
PLOTAD   I2   8  ADDRESSES OF VARIABLES SUBMITTED TO PLOT OUTPUT
DIAMET   R       MEASURED DATA (REAL):   DIAMETER
TEMP1    R           TEMPERATURE
TEMP2    R
```

```
TEMP3    R
SEEDL    R          SEED LIFT
CRUCL    R          CRUCIBLE LIFT
SEEDR    R          SEED ROTATION
CRUCR    R          CRUCIBLE ROTATION
POWER1   R          OUTPUT POWER (FROM SCR)
POWER2   R
POWER3   R
WEIGHT   R          WEIGHT
DWGHT    R          DIFF. WEIGHT
SEEDP    R          SEED POSITION
CRUCP    R          CRUCIBLE POSITION
BASTMP   R          BASE TEMPERATURE
GASPR    R          GAS PRESSURE
PWRIN1   R          POWER INPUT (TO SCR)
PWRIN2   R
PWRIN3   R
LENGTH   R          LENGTH GROWN
ADJDW    R          ADJUSTED DIFF. WEIGHT
REALDT   R  22   MEASURED DATA ARRAY (REAL)
RAMPNG   I1      NUMBER OF VARIABLES RAMPING
STDIAM   I2      CURRENT SETPOINT: DIAMETER
STTMP1   I2          TEMPERATURE
STTMP2   I2
STTMP3   I2
SETSL    I2          SEED LIFT
SETCL    I2          CRUCIBLE LIFT
SETSR    I2          SEED ROTATION
SETCR    I2          CRUCIBLE ROTATION
STPWRL   I2          POWER LIMIT
SETPT0   I2   9  CURRENT SETPOINT ARRAY (INTEGER)
T1GAIN   I1      TEMP. CNTL. I:   GAIN
T1CNTL   I1          CONTROL BYTE
T1PROP   I2          PROP. MULTIPL.
T1INT    I2          INT. MULTIPL.
T1DIFF   I2          DIFF. MULTIPL.
T1LIM    I2          LIMIT
TEMP1P   I2   6  TEMPERATURE CONTROLLER I
T2GAIN   I1      TEMP. CNTL. II:  GAIN
T2CNTL   I1          CONTROL BYTE
T2PROP   I2          PROP. MULTIPL.
T2INT    I2          INT. MULTIPL.
T2DIFF   I2          DIFF. MULTIPL.
T2LIM    I2          LIMIT
TEMP2P   I2   6  TEMPERATURE CONTROLLER II
T3GAIN   I1      TEMP. CNTL. III: GAIN
T3CNTL   I1          CONTROL BYTE
T3PROP   I2          PROP. MULTIPL.
T3INT    I2          INT. MULTIPL.
T3DIFF   I2          DIFF. MULTIPL.
```

T2LIM	I2		LIMIT
TEMP3P	I2	6	TEMPERATURE CONTROLLER III
TEST	I1		TEST MODE FLAG
SLGAIN	I1		SEED LIFT CNTL.: GAIN
SLCNTL	I1		CONTROL BYTE
SLPROP	I2		PROP. MULTIPL.
SLINT	I2		INT. MULTIPL.
SLDIFF	I2		DIFF. MULTIPL.
SLLIM	I2		LIMIT
SEEDLP	I2	6	SEED LIFT CONTROLLER
CLGAIN	I1		CRUC. LIFT CNTL.: GAIN
CLCNTL	I1		CONTROL BYTE
CLPROP	I2		PROP. MULTIPL.
CLINT	I2		INT. MULTIPL.
CLDIFF	I2		DIFF. MULTIPL.
CLLIM	I2		LIMIT
CRUCLP	I2	6	CRUC. LIFT CONTROLLER
SRGAIN	I1		SEED ROT. CNTL.: GAIN
SRCNTL	I1		CONTROL BYTE
SRPROP	I2		PROP. MULTIPL.
SRINT	I2		INT. MULTIPL.
SRDIFF	I2		DIFF. MULTIPL.
SRLIM	I2		LIMIT
SEEDRP	I2	6	SEED ROT CONTROLLER
CRGAIN	I1		CRUC. ROT. CNTL.: GAIN
CRCNTL	I1		CONTROL BYTE
CRPROP	I2		PROP. MULTIPL.
CRINT	I2		INT. MULTIPL.
CRDIFF	I2		DIFF. MULTIPL.
CRLIM	I2		LIMIT
CRUCRP	I2	6	CRUC. ROT. CONTROLLER
DUMPIN	I1		INTERVAL FOR DATA DUMPS
DUMPFL	I1		DUMP FLAG
XTLSHP	R		CRYSTAL SHAPE PARAMETER
SLTHET	I2		THETA VALUES: SEED LIFT
CLTHET	I2		CRUCIBLE LIFT
SRTHET	I2		SEED ROTATION
CRTHET	I2		CRUCIBLE ROTATION
CMMDEX	T		TD FOR CMMDEX
MEASDO	T		TD FOR MEASDO
CMFINP	T		TD FOR CMFINP
CMFOUT	T		TD FOR CMFOUT
DSKOUT	T		TD FOR DISKOUT
DIACNT	T		TD FOR DIACNT
ANACNT	T		TD FOR ANACNT
ALARMF	I1		ALARM TIMER INTERRUPT FLAG
TIME	I2		SYSTEM TIME (INTEGER)
DIFFTM	I2		DIFFERENTIAL TIME FOR MACRO EXECUTION
DTINTV	I1		DATA FILE UPDATING INTERVAL
TIMSET	I2		SETPOINT FOR ALARM TIMER (MACRO EXECUTION)

IOFLAG	I1	4	I/O FLAG ARRAY
ITEMP1	I2		MEASURED DATA (INTEGER): TEMPERATURE
ITEMP2	I2		
ITEMP3	I2		
ISEEDL	I2		SEED LIFT SPEED
ICRUCL	I2		CRUCIBLE LIFT SPEED
ISEEDR	I2		SEED ROTATION
ICRUCR	I2		CRUCIBLE ROTATION
IPOUT1	I2		POWER OUTPUT (FROM SCR)
IPOUT2	I2		
IPOUT3	I2		
IWEIGH	I2		WEIGHT
IDWGHT	I2		DIFF. WEIGHT
ISEEDP	I2		SEED POSITION
ICRUCP	I2		CRUCIBLE POSITION
IBASET	I2		BASE TEMPERATURE
IGASPR	I2		GAS PRESSURE
CONTAC	I2		CONTACT
ANALOG	I2	8	EIGHT SPARE ANALOG CHANNELS (INTEGER)
ANADAT	I2	25	COMPLETE ARRAY OF ANALOG DATA (INTEGER)
IDIAMT	I2		CRYSTAL DIAMETER (INTEGER)
ILENGT	I2		LENGTH GROWN (INTEGER)
SCADIA	R		SCALING FACTORS: DIAMETER
SCATMP	R	3	TEMPERATURES
SCAMOT	R	4	MOTORS
SCAPWO	R	3	POWER OUTPUT
SCAWGT	R		WEIGHT
SCADWT	R		DIFFERENTIAL WEIGHT
SCAPOS	R	2	POSITION
SCABST	R		BASE TEMPERATURE
SCAGAS	R		GAS PRESSURE
SCAPWR	R		POWER INPUT AND LIMIT
SCALE	R	18	ARRAY OF SCALING FACTORS
CDIASQ	R		SQUARE OF CRUCIBLE DIAMETER
SDIASQ	R		SQUARE OF SEED DIAMETER
OXWGHT	R		BORIC OXIDE WEIGHT
RHOXTL	R		DENSITY: CRYSTAL
RHOMLT	R		MELT
RHOOXI	R		OXIDE
SCRUCP	I2		SETPOINT FOR CRUC. POSITION (INTEGER)
ZEROWT	I2		WEIGHT ZEROING OFFSET
GROWTH	R		ACTUAL GROWTH RATE
ALPHA	R		CORRECTION FACTOR FOR GROWTH RATE
DIATAB	R	64	DIAMETER SQUARES TABLE
IHEIGH	I2		MELT HEIGHT (SCALED AS LENGTH)
OLDLEN	I2		LENGTH AT LAST SLICE BOUNDARY
DIFFLG	I2		HEIGHT OF CURRENT SLICE
RDWGHT	R		(ADJUSTED) DIFFERENTIAL WEIGHT (FLOATING-POINT)
RHOOXA	R		ADJUSTED OXIDE DENSITY
DIA1SQ	R		SQUARE OF DIAMETER AT OXIDE SURFACE

```
DIA2SQ   R       SQUARE OF DIAMETER AT MELT SURFACE
HEIGHT   R       BORIC OXIDE HEIGHT IN CRUCIBLE (REAL)
RCRSET   R       CRUCIBLE POSITION SETPOINT (REAL)
VOLSUM   R       SUM OF VOLUMES IN CURRENT SLICE (UNSCALED)
OXIVOL   R       VOLUME OF BORIC OXIDE MELT
CORRVL   R       OXIDE VOLUME CORRECTION
BETA     R       CORRECTION FACTOR
RDLIFT   R       SEED - CRUCIBLE LIFT SPEEDS
RLNGTH   R       UNSCALED LENGTH
PRLNGT   R       UNSCALED LENGTH DURING PREVIOUS PASS
INICRP   R       CRUCIBLE POSITION AT RESET
ADJLEN   R       LENGTH ADJUSTMENT PARAMETER
DIASTA   I1      DIAMETER CONTROLLER STATUS
LOOPCT   I1      LOOP COUNTER LOCATION
OXOVFL   I1      OXIDE HEIGHT OVERFLOW FLAG
MTEMP1   I2      PRIMARY MEASURED DATA (INTEGER): TEMPERATURE
MTEMP2   I2
MTEMP3   I2
MSEEDL   I2          SEED LIFT SPEED
MCRUCL   I2          CRUCIBLE LIFT SPEED
MSEEDR   I2          SEED ROTATION
MCRUCR   I2          CRUCIBLE ROTATION
MPOUT1   I2          POWER OUTPUT (FROM SCR)
MPOUT2   I2
MPOUT3   I2
MWEIGH   I2          WEIGHT
MDWGHT   I2          DIFF. WEIGHT
MSEEDP   I2          SEED POSITION
MCRUCP   I2          CRUCIBLE POSITION
MBASET   I2          BASE TEMPERATURE
MGASPR   I2          GAS PRESSURE
MCONTC   I2          CONTACT
MANALG   I2 25 ANALOG DATA INPUT ARRAY
```

Appendix 11.3: Variable Addresses for CGCS Versions 2.0 - 2.4

V2.0	V2.1	V2.2	V2.3	V2.4				
1FF6	1FF6	1FF6	1FF6	1FF6	ZERO	I2		LOCATION HOLDING ZERO
2136	2136	2136	2136	2136	DISKIO	T		TD FOR TASK DISKIO
214A	214A	214A	214A	214A	RQTHDI	T		TD FOR TASK RQTHDI
215E	215E	215E	215E	215E	RQTHDO	T		TD FOR TASK RQTHDO
2172	2172	2172	2172	2172	RQLOAD	T		TD FOR TASK RQLOAD
2186	2186	2186	2186	2186	COMINT	T		TD FOR TASK COMINT = RXIROM
2186	2186	2186	2186	2186	RXIROM	T		
2800	2800	2800	2800	2800	PWR1IN	I2		INTEGER: POWER INPUT (TO SCR MODULE)
2802	2802	2802	2802	2802	PWR2IN	I2		
2804	2804	2804	2804	2804	PWR3IN	I2		
2806	2806	2806	2806	2806	SEEDLO	I2		INTEGER: SEED LIFT OUTPUT
2808	2808	2808	2808	2808	CRUCLO	I2		INTEGER: CRUCIBLE LIFT OUTPUT
280A	280A	280A	280A	280A	SEEDRO	I2		INTEGER: SEED ROTATION OUTPUT
280C	280C	280C	280C	280C	CRUCRO	I2		INTEGER: CRUCIBLE ROTATION OUTPUT
280E	280E	280E	280E	280E	PLOTDT	I2	8	INTEGER: OUTPUT TO CHART RECORDER
2800	2800	2800	2800	2800	ANAOUT	I2	16	ARRAY OF INTEGER OUTPUT DATA
2820	2820	2820	2820	2820	ANIPAR	I1	52	PARAMETER ARRAY FOR ANALOG INPUT
2854	2854	2854	2854	2854	ANOMLY	R	2	ANOMALY CORRECTION PARAMETERS (REAL)
285C	285C	285C	285C	285C	ANOPAR	I1	17	PARAMETER ARRAY FOR ANALOG OUTPUT
286D	286D	286D	286D	286D	OFFST1	R		TEMPERATURE OFFSET - HEATER TEMP I
2871	2871	2871	2871	2871	OFFST2	R		HEATER TEMPERATURE II
2875	2875	2875	2875	2875	OFFST3	R		HEATER TEMPERATURE III
2879	2879	2879	2879	2879	OFFSTB	R		BASE TEMPERATURE
....	287D	287D	287D	287D	RANGT1	R		TEMP. CHART RECORDER OUTPUT RANGE - I
....	2881	2881	2881	2881	RANGT2	R		HEATER TEMPERATURE II
....	2885	2885	2885	2885	RANGT3	R		HEATER TEMPERATURE III
....	2889	2889	2889	2889	RANGTB	R		BASE TEMPERATURE
287D	288D	288D	288D	288D	EXTMP1	I2		EXPANDED HEATER I TEMPERATURE
287F	288F	288F	288F	288F	EXTMP2	I2		HEATER II
2881	2891	2891	2891	2891	EXTMP3	I2		HEATER III
2883	2893	2893	2893	2893	EXTMPB	I2		BASE
2885	2895	2895	2895	2895	DIAERR	I2		DIAMETER ERROR (FOR PLOT)
2887	2897	2897	2897	2897	CRPERR	I2		CRUCIBLE POSITION ERROR (FOR PLOT)
2889	2899	2899	2899	2899	GRRATE	I2		GROWTH RATE (FOR PLOT)
2888	2898	2898	2898	2898	DUMMY	I2	8	EIGHT DUMMY LOCATIONS
2898	28AB	28AB	28AB	28AB	CNDCNT	I1		COUNTER FOR CONDITIONAL COMMANDS
2880	28C0	28C0	28C0	28C0	CONLIM	I2		INTEGER: LIMIT VALUE FOR CONTACT DEV.
2882	28C2	28C2	28C2	28C2	COGAIN	I1		CRUCIBLE LIFT CONTROLLER ARRAY: GAIN
2883	28C3	28C3	28C3	28C3	COCNTL	I1		CONTROL BYTE
2886	28C6	28C6	28C6	28C6	COPROP	I2		PROP. MULTIPL.
2888	28C8	28C8	28C8	28C8	COINT	I2		INT. MULTIP.
288A	28CA	28CA	28CA	28CA	CODIFF	I2		DIFF. MULTIP.
288C	28CC	28CC	28CC	28CC	COLIM	I2		LIMIT
2882	28C2	28C2	28C2	28C2	CRUCOP	I2	6	CRUCIBLE LIFT CONTROLLER ARRAY
288E	28CE	28CE	28CE	28CE	C1GAIN	I1		AUXILIARY CRUC. LIFT CONTROLLER: GAIN
288F	28CF	28CF	28CF	28CF	C1CNTL	I1		CONTROL BYTE

28C2	28D2	28D2	28D2	28D2	C1PROP	I2		PROP. MULTIPL.
28C4	28D4	28D4	28D4	28D4	C1INT	I2		INT. MULTIPL.
28C6	28D6	28D6	28D6	28D6	C1DIFF	I2		DIFF. MULTIPL.
28C8	28D8	28D8	28D8	28D8	C1LIM	I2		LIMIT
28BE	28CE	28CE	28CE	28CE	CRUC1P	I2	6	AUXILIARY CRUCIBLE LIFT CONTROLLER
28F6	2906	2906	2906	2906	GAIN10	I1		MAIN DIAM. CNTL. I GAIN
28F7	2907	2907	2907	2907	CNTL10	I1		CONTROL BYTE
28FA	290A	290A	290A	290A	PROP10	I2		PROP. MULTIPL.
28FC	290C	290C	290C	290C	INT10	I2		INT. MULTIPL.
28FE	290E	290E	290E	290E	DIFF10	I2		DIFF. MULTIPL.
2900	2910	2910	2910	2910	LIM10	I2		LIMIT
28F6	2906	2906	2906	2906	DIA10P	I2	6	MAIN DIAMETER CONTROLLER I
2902	2912	2912	2912	2912	GAIN11	I1		AUX. DIAM. CNTL. I GAIN
2903	2913	2913	2913	2913	CNTL11	I1		CONTROL BYTE
2906	2916	2916	2916	2916	PROP11	I2		PROP. MULTIPL.
2908	2918	2918	2918	2918	INT11	I2		INT. MULTIPL.
290A	291A	291A	291A	291A	DIFF11	I2		DIFF. MULTIPL.
290C	291C	291C	291C	291C	LIM11	I2		LIMIT
2902	2912	2912	2912	2912	DIA11P	I2	6	AUXILIARY DIAMETER CONTROLLER I
290E	291E	291E	291E	291E	GAIN20	I1		MAIN DIAM. CNTL. II GAIN
290F	291F	291F	291F	291F	CNTL20	I1		CONTROL BYTE
2912	2922	2922	2922	2922	PROP20	I2		PROP. MULTIPL.
2914	2924	2924	2924	2924	INT20	I2		INT. MULTIPL.
2916	2926	2926	2926	2926	DIFF20	I2		DIFF. MULTIPL.
2918	2928	2928	2928	2928	LIM20	I2		LIMIT
290E	291E	291E	291E	291E	DIA20P	I2	6	MAIN DIAMETER CONTROLLER II
291A	292A	292A	292A	292A	GAIN21	I1		AUX. DIAM. CNTL. II GAIN
291B	292B	292B	292B	292B	CNTL21	I1		CONTROL BYTE
291E	292E	292E	292E	292E	PROP21	I2		PROP. MULTIPL.
2920	2930	2930	2930	2930	INT21	I2		INT. MULTIPL.
2922	2932	2932	2932	2932	DIFF21	I2		DIFF. MULTIPL.
2924	2934	2934	2934	2934	LIM21	I2		LIMIT
291A	292A	292A	292A	292A	DIA21P	I2	6	AUXILIARY DIAMETER CONTROLLER II
2926	2936	2936	2936	2936	GAIN30	I1		MAIN DIAM. CNTL. III GAIN
2927	2937	2937	2937	2937	CNTL30	I1		CONTROL BYTE
292A	293A	293A	293A	293A	PROP30	I2		PROP. MULTIPL.
292C	293C	293C	293C	293C	INT30	I2		INT. MULTIPL.
292E	293E	293E	293E	293E	DIFF30	I2		DIFF. MULTIPL.
2930	2940	2940	2940	2940	LIM30	I2		LIMIT
2926	2936	2936	2936	2936	DIA30P	I2	6	MAIN DIAMETER CONTROLLER III
2932	2942	2942	2942	2942	GAIN31	I1		AUX. DIAM. CNTL. III GAIN
2933	2943	2943	2943	2943	CNTL31	I1		CONTROL BYTE
2936	2946	2946	2946	2946	PROP31	I2		PROP. MULTIPL.
2938	2948	2948	2948	2948	INT31	I2		INT. MULTIPL.
293A	294A	294A	294A	294A	DIFF31	I2		DIFF. MULTIPL.
293C	294C	294C	294C	294C	LIM31	I2		LIMIT
2932	2942	2942	2942	2942	DIA31P	I2	6	AUXILIARY DIAMETER CONTROLLER III
2985	2991	2991	2991	2991	INTRVL	I2		INTERVAL FOR MEASUREMENT DATA OUTPUT
298F	299B	299B	299B	299B	PLOTAD	I2	8	ADDRESSES OF CHART RECORDER OUTPUT
299F	29AB	29AB	29AB	29AB	DIAMET	R		MEASURED DATA (REAL): DIAMETER

29A3	29AF	29AF	29AF	29AF	TEMP1	R		TEMPERATURE
29A7	29B3	29B3	29B3	29B3	TEMP2	R		
29AB	29B7	29B7	29B7	29B7	TEMP3	R		
29AF	29BB	29BB	29BB	29BB	SEEDL	R		SEED LIFT
29B3	29BF	29BF	29BF	29BF	CRUCL	R		CRUCIBLE LIFT
29B7	29C3	29C3	29C3	29C3	SEEDR	R		SEED ROTATION
29BB	29C7	29C7	29C7	29C7	CRUCR	R		CRUCIBLE ROTATION
29BF	29CB	29CB	29CB	29CB	POWER1	R		OUTPUT POWER (FROM SCR)
29C3	29CF	29CF	29CF	29CF	POWER2	R		
29C7	29D3	29D3	29D3	29D3	POWER3	R		
29CB	29D7	29D7	29D7	29D7	WEIGHT	R		WEIGHT
29CF	29DB	29DB	29DB	29DB	DWGHT	R		DIFFERENTIAL WEIGHT
29D3	29DF	29DF	29DF	29DF	SEEDP	R		SEED POSITION
29D7	29E3	29E3	29E3	29E3	CRUCP	R		CRUCIBLE POSITION
29DB	29E7	29E7	29E7	29E7	BASTMP	R		BASE TEMPERATURE
29DF	29EB	29EB	29EB	29EB	GASPR	R		GAS PRESSURE
29E3	29EF	29EF	29EF	29EF	PWRIN1	R		POWER INPUT (TO SCR)
29E7	29F3	29F3	29F3	29F3	PWRIN2	R		
29EB	29F7	29F7	29F7	29F7	PWRIN3	R		
29EF	29FB	29FB	29FB	29FB	LENGTH	R		LENGTH GROWN
29F3	29FF	29FF	29FF	29FF	ADJDW	R		ADJUSTED DIFF. WEIGHT
299F	29AB	29AB	29AB	29AB	REALDT	R	22	MEASURED DATA ARRAY (REAL)
29FD	2A09	2A09	2A09	2A09	RAMPNG	I1		NUMBER OF VARIABLES RAMPING
2B9E	2BAA	2BAA	2BAA	2BAA	STDIAM	I2		CURRENT SETPOINT DIAMETER
2BA0	2BAC	2BAC	2BAC	2BAC	STTMP1	I2		TEMPERATURE
2BA2	2BAE	2BAE	2BAE	2BAE	STTMP2	I2		
2BA4	2BB0	2BB0	2BB0	2BB0	STTMP3	I2		
2BA6	2BB2	2BB2	2BB2	2BB2	SETSL	I2		SEED LIFT
2BA8	2BB4	2BB4	2BB4	2BB4	SETCL	I2		CRUCIBLE LIFT
2BAA	2BB6	2BB6	2BB6	2BB6	SETSR	I2		SEED ROTATION
2BAC	2BB8	2BB8	2BB8	2BB8	SETCR	I2		CRUCIBLE ROTATION
2BAE	2BBA	2BBA	2BBA	2BBA	STPWRL	I2		POWER LIMIT
2B9E	2BAA	2BAA	2BAA	2BAA	SETPTO	I2	9	CURRENT SETPOINT ARRAY (INTEGER)
2BD1	2BDD	2BDD	2BDD	2BDD	T1GAIN	I1		TEMP. CNTL. I GAIN
2BD2	2BDE	2BDE	2BDE	2BDE	T1CNTL	I1		CONTROL BYTE
2BD5	2BE1	2BE1	2BE1	2BE1	T1PROP	I2		PROP. MULTIPL.
2BD7	2BE3	2BE3	2BE3	2BE3	T1INT	I2		INT. MULTIPL.
2BD9	2BE5	2BE5	2BE5	2BE5	T1DIFF	I2		DIFF. MULTIPL.
2BDB	2BE7	2BE7	2BE7	2BE7	T1LIM	I2		LIMIT
2BD1	2BDD	2BDD	2BDD	2BDD	TEMP1P	I2	6	TEMPERATURE CONTROLLER I
2BDD	2BE9	2BE9	2BE9	2BE9	T2GAIN	I1		TEMP. CNTL. II GAIN
2BDE	2BEA	2BEA	2BEA	2BEA	T2CNTL	I1		CONTROL BYTE
2BE1	2BED	2BED	2BED	2BED	T2PROP	I2		PROP. MULTIPL.
2BE3	2BEF	2BEF	2BEF	2BEF	T2INT	I2		INT. MULTIPL.
2BE5	2BF1	2BF1	2BF1	2BF1	T2DIFF	I2		DIFF. MULTIPL.
2BE7	2BF3	2BF3	2BF3	2BF3	T2LIM	I2		LIMIT
2BDD	2BE9	2BE9	2BE9	2BE9	TEMP2P	I2	6	TEMPERATURE CONTROLLER II
2BE9	2BF5	2BF5	2BF5	2BF5	T3GAIN	I1		TEMP. CNTL. III GAIN
2BEA	2BF6	2BF6	2BF6	2BF6	T3CNTL	I1		CONTROL BYTE
2BED	2BF9	2BF9	2BF9	2BF9	T3PROP	I2		PROP. MULTIPL.

2BEF	2BFB	2BFB	2BFB	2BFB	T3INT	I2		INT. MULTIPL.
2BF1	2BFD	2BFD	2BFD	2BFD	T3DIFF	I2		DIFF. MULTIPL.
2BF3	2BFF	2BFF	2BFF	2BFF	T2LIM	I2		LIMIT
2BE9	2BF5	2BF5	2BF5	2BF5	TEMP3P	I2	6	TEMPERATURE CONTROLLER III
2BF5	2C01	2C01	2C01	2C01	TEST	I1		TEST MODE FLAG
2C00	2C0C	2C0C	2C0C	2C0C	SLGAIN	I1		SEED LIFT CNTL.: GAIN
2C01	2C0D	2C0D	2C0D	2C0D	SLCNTL	I1		CNTL
2C04	2C10	2C10	2C10	2C10	SLPROP	I2		PROP. MULTIPLIER
2C06	2C12	2C12	2C12	2C12	SLINT	I2		INT. MULTIPL.
2C08	2C14	2C14	2C14	2C14	SLDIFF	I2		DIFF. MULTIPL.
2C0A	2C16	2C16	2C16	2C16	SLLIM	I2		LIMIT
2C00	2C0C	2C0C	2C0C	2C0C	SEEDLP	I2	6	SEED LIFT CONTROLLER
2C0C	2C18	2C18	2C18	2C18	CLGAIN	I1		CRUC LIFT CNTL.: GAIN
2C0D	2C19	2C19	2C19	2C19	CLCNTL	I1		CNTL
2C10	2C1C	2C1C	2C1C	2C1C	CLPROP	I2		PROP. MULTIPLIER
2C12	2C1E	2C1E	2C1E	2C1E	CLINT	I2		INT. MULTIPL.
2C14	2C20	2C20	2C20	2C20	CLDIFF	I2		DIFF. MULTIPL.
2C16	2C22	2C22	2C22	2C22	CLLIM	I2		LIMIT
2C0C	2C18	2C18	2C18	2C18	CRUCLP	I2	6	CRUC LIFT CONTROLLER
2C18	2C24	2C24	2C24	2C24	SRGAIN	I1		SEED ROT CNTL.: GAIN
2C19	2C25	2C25	2C25	2C25	SRCNTL	I1		CNTL
2C1C	2C28	2C28	2C28	2C28	SRPROP	I2		PROP. MULTIPLIER
2C1E	2C2A	2C2A	2C2A	2C2A	SRINT	I2		INT. MULTIPL.
2C20	2C2C	2C2C	2C2C	2C2C	SRDIFF	I2		DIFF. MULTIPL.
2C22	2C2E	2C2E	2C2E	2C2E	SRLIM	I2		LIMIT
2C18	2C24	2C24	2C24	2C24	SEEDRP	I2	6	SEED ROT CONTROLLER
2C24	2C30	2C30	2C30	2C30	CRGAIN	I1		CRUC ROT CNTL.: GAIN
2C25	2C31	2C31	2C31	2C31	CRCNTL	I1		CNTL
2C28	2C34	2C34	2C34	2C34	CRPROP	I2		PROP. MULTIPLIER
2C2A	2C36	2C36	2C36	2C36	CRINT	I2		INT. MULTIPL.
2C2C	2C38	2C38	2C38	2C38	CRDIFF	I2		DIFF. MULTIPL.
2C2E	2C3A	2C3A	2C3A	2C3A	CRLIM	I2		LIMIT
2C24	2C30	2C30	2C30	2C30	CRUCRP	I2	6	CRUC ROT CONTROLLER
2C30	2C3C	2C3C	2C3C	2C3C	DUMPIN	I1		INTERVAL FOR DATA DUMPS
2C31	2C3D	2C3D	2C3D	2C3D	DUMPFL	I1		DUMP FLAG
----	----	2C3E	2C3E	2C3E	XTLSHP	R		CRYSTAL SHAPE PARAMETER
----	----	----	----	2C42	SLTHET	I2		THETA VALUES: SEED LIFT
----	----	----	----	2C44	CLTHET	I2		CRUCIBLE LIFT
----	----	----	----	2C46	SRTHET	I2		SEED ROTATION
----	----	----	----	2C48	CRTHET	I2		CRUCIBLE ROTATION
3152	3152	3120	3120	3120	CMMDEX	T		TD FOR CMMDEX
317A	317A	3148	3148	3148	MEASDO	T		TD FOR MEASDO
31A2	31A2	3170	3170	3170	CMFINP	T		TD FOR CMFINP
31CA	31CA	3198	3198	3198	CMFOUT	T		TD FOR CMFOUT
31F2	31F2	----	----	----	DISKO0	T		TD FOR DISKO0
3206	3206	----	----	----	DISKO1	T		TD FOR DISKO1
----	----	31C0	31C0	31C0	DSKOUT	T		TD FOR DSKOUT
322E	322E	31E8	31E8	31E8	DIACNT	T		TD FOR DIACNT
3256	3256	3210	3210	3210	ANACNT	T		TD FOR ANACNT
3387	3387	3341	3341	3341	ALARMF	I1		ALARM TIMER INTERRUPT FLAG

3388	3388	3342	3342	3342	TIME	I2		SYSTEM TIME (INTEGER)
338A	338A	3344	3344	3344	DIFFTM	I2		DIFFERENTIAL TIME FOR MACRO EXECUTION
33A4	33A4	335E	335E	335E	DTINTV	I1		DATA FILE UPDATING INTERVAL
33A6	33A6	3360	3360	3360	TIMSET	I2		SETPOINT FOR ALARM TIMER (MACRO EXE.)
3443	3443	33FD	33FD	33FD	IOFLAG	I1	4	I/O FLAG ARRAY
34FD	3581	353B	353B	353B	ITEMP1	I2		MEASURED DATA (INTEGER): TEMPERATURE
34FF	3583	353D	353D	353D	ITEMP2	I2		
3501	3585	353F	353F	353F	ITEMP3	I2		
3503	3587	3541	3541	3541	ISEEDL	I2		SEED LIFT SPEED
3505	3589	3543	3543	3543	ICRUCL	I2		CRUCIBLE LIFT SPEED
3507	358B	3545	3545	3545	ISEEDR	I2		SEED ROTATION
3509	358D	3547	3547	3547	ICRUCR	I2		CRUCIBLE ROTATION
350B	358F	3549	3549	3549	IPOUT1	I2		POWER OUTPUT (FROM SCR)
350D	3591	354B	354B	354B	IPOUT2	I2		
350F	3593	354D	354D	354D	IPOUT3	I2		
3511	3595	354F	354F	354F	IWEIGH	I2		WEIGHT
3513	3597	3551	3551	3551	IDWGHT	I2		DIFF. WEIGHT
3515	3599	3553	3553	3553	ISEEDP	I2		SEED POSITION
3517	359B	3555	3555	3555	ICRUCP	I2		CRUCIBLE POSITION
3519	359D	3557	3557	3557	IBASET	I2		BASE TEMPERATURE
351B	359F	3559	3559	3559	IGASPR	I2		GAS PRESSURE
351D	35A1	355B	355B	355B	CONTAC	I2		CONTACT
351F	35A3	355D	355D	355D	ANALOG	I2	8	EIGHT SPARE ANALOG CHANNELS (INTEGER)
34FD	3581	353B	353B	353B	ANADAT	I2	25	COMPLETE ARRAY OF ANALOG DATA (INT.)
352F	35B3	356D	356D	356D	IDIAMT	I2		CRYSTAL DIAMETER (INTEGER)
3531	35B5	356F	356F	356F	ILENGT	I2		LENGTH GROWN (INTEGER)
3533	35B7	3571	3571	3571	SCADIA	R		SCALING FACTORS: DIAMETER
3537	35BB	3575	3575	3575	SCATMP	R	3	TEMPERATURES
3543	35C7	3581	3581	3581	SCAMOT	R	4	MOTORS
3553	35D7	3591	3591	3591	SCAPWO	R	3	POWER OUTPUT
355F	35E3	359D	359D	359D	SCAWGT	R		WEIGHT
3563	35E7	35A1	35A1	35A1	SCADWT	R		DIFFERENTIAL WEIGHT
3567	35EB	35A5	35A5	35A5	SCAPOS	R	2	POSITION
356F	35F3	35AD	35AD	35AD	SCABST	R		BASE TEMPERATURE
3573	35F7	35B1	35B1	35B1	SCAGAS	R		GAS PRESSURE
3577	35FB	35B5	35B5	35B5	SCAPWR	R		POWER INPUT AND LIMIT
3533	35B7	3571	3571	3571	SCALE	R	18	ARRAY OF SCALING FACTORS
357B	35FF	35B9	35B9	35B9	CDIASQ	R		SQUARE OF CRUCIBLE DIAMETER
357F	3603	35BD	35BD	35BD	SDIASQ	R		SQUARE OF SEED DIAMETER
3583	3607	35C1	35C1	35C1	OXWGHT	R		BORIC OXIDE WEIGHT
3587	360B	35C5	35C5	35C5	RHOXTL	R		DENSITY: CRYSTAL
358B	360F	35C9	35C9	35C9	RHOMLT	R		MELT
358F	3613	35CD	35CD	35CD	RHOOXI	R		OXIDE
3593	3617	35D1	35D1	35D1	SCRUCP	I2		SETPOINT FOR CRUC. POSITION (INTEGER)
3595	3619	35D3	35D3	35D3	ZEROWT	I2		WEIGHT ZEROING VALUE
3597	361B	35D5	35D5	35D5	GROWTH	R		ACTUAL GROWTH RATE
3598	....	....	....	....	RHEIGH	R		MELT HEIGHT IN CRUCIBLE (REAL)
359F	....	....	....	....	INICRP	R		INITIAL CRUCIBLE POSITION (AT RESET)
35A3	....	....	....	....	INIWGT	R		INITIAL CRYSTAL WEIGHT (AT RESET)
35A7	....	....	....	....	RCRSET	R		SETPOINT FOR CRUC. POSITION (REAL)

35AB	----	----	----	----	ADJLEN	R		LENGTH ADJUSTMENT (REAL)
35AF	----	----	----	----	DIATAB	R	64	TABLE OF CRYSTAL DIAMETERS
36B1	----	----	----	----	DIASTA	I1		DIAMETER CONTROLLER STATUS
36B6	----	----	----	----	HEIGHT	I2		MELT HEIGHT (SCALED AS LENGTH)
36B8	----	----	----	----	RHOOXA	R		ADJUSTED OXIDE DENSITY
36BC	----	----	----	----	DIA1SQ	R		SQUARE OF DIAMETER AT OXIDE SURFACE
36C0	----	----	----	----	DIA2SQ	R		SQUARE OF DIAMETER AT MELT SURFACE
36C6	----	----	----	----	POINTS	I2		NUMBER OF DATA POINTS IN SUMMATION
36C8	----	----	----	----	DIA1SM	R		SUM OF DIAMETER SQ. AT OXIDE SURFACE
36CC	----	----	----	----	DIA2SM	R		SUM OF DIAMETER SQ. AT MELT SURFACE
36D4	----	----	----	----	STEP	R		STEP FOR MELT HEIGHT EVALUATION
----	----	----	----	----	GROWTH	R		ACTUAL GROWTH RATE
36D8	----	----	----	----	RSEEDL	R		SEED LIFT SPEED (FLOATING-POINT)
36DC	----	----	----	----	RCRUCL	R		CRUCIBLE LIFT SPEED (FLOATING-POINT)
----	361F	35D9	35D9	35D9	ALPHA	R		CORRECTION FACTOR FOR GROWTH RATE
----	----	35DD	35DD	35DD	DIATAB	R	64	DIAMETER SQUARES TABLE
----	3623	36DD	36DD	36DD	IHEIGH	I2		MELT HEIGHT (SCALED AS LENGTH)
----	3625	36DF	36DF	36DF	OLDLEN	I2		LENGTH AT LAST SLICE BOUNDARY
----	3627	36E1	36E1	36E1	DIFFLG	I2		HEIGHT OF CURRENT SLICE
----	----	----	36E5	36E5	RDWGHT	R		(ADJUSTED) DIFFERENTIAL WEIGHT
----	362F	36E9	36E9	36E9	RHOOXA	R		ADJUSTED OXIDE DENSITY
----	3633	36ED	36ED	36ED	DIA1SQ	R		SQUARE OF DIAMETER AT OXIDE SURFACE
----	3637	36F1	36F1	36F1	DIA2SQ	R		SQUARE OF DIAMETER AT MELT SURFACE
----	363B	36F5	36F5	36F5	HEIGHT	R		BORIC OXIDE HEIGHT IN CRUCIBLE (REAL)
----	----	36F9	36F9	36F9	RCRSET	R		CRUCIBLE POSITION SETPOINT (REAL)
----	363F	36FD	36FD	36FD	VOLSUM	R		SUM OF VOLUMES IN CURRENT SLICE
----	3643	3701	3701	3701	OXIVOL	R		VOLUME OF BORIC OXIDE MELT
----	----	3705	3705	3705	CORRVL	R		OXIDE VOLUME CORRECTION
----	----	3709	3709	3709	BETA	R		CORRECTION FACTOR
----	3647	370D	370D	370D	RDLIFT	R		SEED - CRUCIBLE LIFT SPEEDS
----	364B	3711	3711	3711	RLNGTH	R		UNSCALED LENGTH
----	364F	3715	3715	3715	PRLNGT	R		UNSCALED LENGTH DURING PREVIOUS PASS
----	3657	371D	371D	3719	INICRP	R		CRUCIBLE POSITION AT RESET
----	365B	----	----	----	RCRSET	R		CRUCIBLE POSITION SETPOINT (REAL)
----	365F	3721	3721	371D	ADJLEN	R		LENGTH ADJUSTMENT PARAMETER
----	3663	----	----	----	DIATAB	R	64	DIAMETER SQUARES TABLE
----	3765	3727	3727	3723	DIASTA	I1		DIAMETER CONTROLLER STATUS
----	3766	3728	3728	3724	LOOPCT	I1		LOOP COUNTER LOCATION
----	----	3729	3729	3725	OXOVFL	I1		OXIDE HEIGHT OVERFLOW FLAG
----	----	----	----	3726	SHELP	R		SHAPE AUXILIARY LOCATIONS
----	----	----	----	372A	SHELP1	R		
----	----	----	----	372E	SHELP2	R		
4164	41EB	4133	4179	4183	MTEMP1	I2		ANALOG MEASUREMENT DATA: TEMPERATURE I
4166	41ED	4135	417B	4185	MTEMP2	I2		TEMPERATURE II
4168	41EF	4137	417D	4187	MTEMP3	I2		TEMPERATURE III
416A	41F1	4139	417F	4189	MSEEDL	I2		SEED LIFT
416C	41F3	413B	4181	418B	MCRUCL	I2		CRUCIBLE LIFT
416E	41F5	413D	4183	418D	MSEEDR	I2		SEED ROTATION
4170	41F7	413F	4185	418F	MCRUCR	I2		CRUCIBLE ROTATION
4172	41F9	4141	4187	4191	MPOUT1	I2		POWER OUTPUT (FROM SCR)

4174	41FB	4143	4189	4193	MPOUT2	I2		
4176	41FD	4145	418B	4195	MPOUT3	I2		
4178	41FF	4147	418D	4197	MWEIGH	I2		WEIGHT
417A	4201	4149	418F	4199	MDWGHT	I2		DIFFERENTIAL WEIGHT
417C	4203	414B	4191	419B	MSEEDP	I2		SEED POSITION
417E	4205	414D	4193	419D	MCRUCP	I2		CRUCIBLE POSITION
4180	4207	414F	4195	419F	MBASET	I2		BASE TEMPERATURE
4182	4209	4151	4197	41A1	MGASPR	I2		GAS PRESSURE
4184	420B	4153	4199	41A3	MCONTC	I2		CONTACT DEVICE
4162	41E9	4131	4177	4181	MANALG	I2	25	ANALOG DATA INPUT ARRAY

Appendix 12: CGCS File Formats

Appendix 12.1: Variable Name File CZONAM.Vmn

The file name extension of the Variable Name file has to hold the major and minor version codes of the CGCS system release to which the file refers. (CZONAM.V24 is, for example, the Variable Name file for CGCS Version 2.4.) The file is built of records of 128 bytes, each of which holds 14 entries of 9 bytes each; each record is terminated by a Carriage-Return - Line Feed pair.

Each entry contains:

Bytes 1 - 6: Variable name (1 - 6 uppercase characters, left justified, right filled with spaces.

Byte 7: Variable type and array size, encoded as (type number + (array size - 1) * 4), where "array size" is the number of array elements (1 to 64). The following "type" values are defined:

type = 0 ... iRMX-80 control structure
type = 1 ... one-byte integer (INTEGER*1)
type = 2 ... two-byte integer (INTEGER*2)
type = 3 ... floating-point number (REAL)

Bytes 8 - 9: (Start) address of the specified variable.

Appendix 12.2: Variable Name Source File

The source file which holds the Variable names and which is eventually converted to a CZONAM file with the utility program CONVAD does not require very strict formatting but must follow the subsequent rules:

(1) Each entry must be held in a separate line in the following order:

(a) Address (in hexadecimal notation, with or without trailing "H").

(b) Variable name (in capitals), 1 to 6 characters.

(c) Variable type number (0 through 3; see Appendix 12.1).

(d) Number of array elements (optional); a missing number is interpreted as "1".

(e) Comment (optional); the comment field should not contain digits lest they could be interpreted as an array size.

(2) Items within a line must be separated from one another by one or more blanks (spaces, TAB characters, etc.).

Appendix 12.3: Macro Command Files

Macro files (and therefore also the Command Output files) are built of records of 16 bytes each. They consist of one header record and an arbitrary number (including zero) of data records.

Header Record:

Bytes 1 - 2:	Zero.
Byte 3:	Minor CGCS Version code.
Byte 4:	Major CGCS Version code.
Bytes 5 - 16:	Reserved.

Data Records:

Bytes 1 - 2:	Relative time of command as unsigned two-byte integer seconds count (0 - 65535).
Byte 3:	Command code byte.
Bytes 4 - 16:	Depend on command code; see below.

Command Codes:

11H Set Diameter
12H Set Heater Temperature #1
13H Set Heater Temperature #2
14H Set Heater Temperature #3
15H Set Seed Lift speed
16H Set Crucible Lift speed
17H Set Seed Rotation speed
18H Set Crucible Rotation speed
19H Set Power Limit

21H Modify Diameter
22H Modify Heater Temperature #1
23H Modify Heater Temperature #2

24H Modify Heater Temperature #3
25H Modify Seed Lift speed
26H Modify Crucible Lift speed
27H Modify Seed Rotation speed
28H Modify Crucible Rotation speed
29H Modify Power Limit

Bytes 4 - 5: New setpoint or setpoint change (INTEGER*2).
Bytes 6 - 9: Transition time in seconds (REAL).
Bytes 10 - 16: Reserved.

30H Macro Command

Bytes 4 - 9: Macro Command name (left justified, right filled with spaces).
Bytes 10 - 16: Reserved.

31H Clear Conditional Macros Unconditionally

Bytes 4 - 16: Reserved.

40H Mode = Monitoring
41H Mode = Manual
42H Mode = Diameter
43H Mode = Diameter/ASC
44H Mode = Automatic

Bytes 4 - 16: Reserved.

70H Reset

Bytes 4 - 5: New weight (INTEGER*2).
Bytes 6 - 7: New length (INTEGER*2).
Bytes 8 - 16: Reserved.

7FH End of Command Record

Bytes 4 - 16: Reserved.

90H Set Variable
A0H Change Variable

Byte 4: Variable type:
2 ... INTEGER*1
4 ... INTEGER*2
6 ... REAL
Bytes 5 - 6: Variable address (INTEGER*2).
Bytes 7 - 10: New setpoint or change value (REAL).
Bytes 11 - 14: Transition time in seconds (REAL).
Bytes 15 - 16: Reserved.

B0H Conditional Command

Byte 4: Variable type + 16 * Relation code #2
+ 64 * Relation code #1, with:
Variable type:
2 ... INTEGER*1
4 ... INTEGER*2
6 ... REAL
Relation code:
1 ... "<"
2 ... "="
3 ... ">"
Bytes 5 - 6: Variable address (INTEGER*2).
Bytes 7 - 10: Comparison value (REAL).
Bytes 11 - 16: Macro Command name.

B1H Clear Conditional Commands Selectively

Byte 4: Reserved.
Bytes 5 - 6: Variable address (INTEGER*2).
Bytes 7 - 16: Reserved.

E0H Assign Plot Channel

Byte 4: Plot channel number (1 - 8)
Bytes 5 - 6: Variable address (INTEGER*2).
Bytes 7 - 16: Reserved.

F2H Debug Continuously
F3H Debug Modify
F4H Debug Resume
F5H Debug Suspend

Byte 4:	Variable type + 16 * Output location, with: Variable type: 1 ... ASCII (1 character) 2 ... INTEGER*1 3 ... one-byte hexadecimal 4 ... INTEGER*2 5 ... two-byte hexadecimal 6 ... REAL 7 ... four-byte hexadecimal Output location: 1 - 4
Bytes 5 - 6:	Variable address (INTEGER*2).
Bytes 7 - 10:	New value.
Bytes 11 - 16:	Reserved.

(Most Debug commands need only part of the information in bytes 4 - 10)

The contents of a command message are identical to those of the corresponding command record bytes 3 through 16.

Appendix 12.4: Data Files

A Data file is made up of records of 128 bytes each. It consists of one Header record and an arbitrary number of Data and Comment records. With the exception of the first two bytes, Data records are built of two-byte words, i.e., 64 words per record. All data are in INTEGER*2 format unless noted otherwise.

Header Record:

Bytes 1 - 8:	Date (8 ASCII characters).
Bytes 9 - 28:	Run Identification (20 ASCII characters).
Bytes 29 - 30:	Record interval (two hexadecimal digits).
Byte 31:	Major CGCS system version code.
Byte 32:	Minor CGCS system version code.
Bytes 33 - 128:	Contents of bytes 1 - 32 repeated three times.

Data Record:

Byte 1:	Always 0.
Byte 2:	Operation Mode (INTEGER*1).
Word 2:	System time.
Word 3:	Length grown.

Word 4:	Temperature #1.	(Measured Data)
Word 5:	Temperature #2.	
Word 6:	Temperature #3.	
Word 7:	Seed Lift.	
Word 8:	Crucible Lift.	
Word 9:	Seed Rotation.	
Word 10:	Crucible Rotation.	
Word 11:	Power Output #1.	
Word 12:	Power Output #2.	
Word 13:	Power Output #3.	
Word 14:	Weight.	
Word 15:	Differential Weight.	
Word 16:	Seed Position.	
Word 17:	Crucible Position.	
Word 18:	Base Temperature.	
Word 19:	Gas Pressure.	
Word 20:	Contact Device.	(Measured Data)
Words 21 - 28:	Eight Spare Analog Input Channels.	
Word 29:	Power Input #1.	(Control Output)
Word 30:	Power Input #2.	
Word 31:	Power Input #3.	(Control Output)
Word 32:	Diameter.	(Current Setpoints)
Word 33:	Temperature #1.	
Word 34:	Temperature #2.	
Word 35:	Temperature #3.	
Word 36:	Seed Lift.	
Word 37:	Crucible Lift.	
Word 38:	Seed Rotation.	
Word 39:	Crucible Rotation.	
Word 40:	Power Limit.	(Current Setpoints)
Word 41:	Diameter.	(Final Setpoints)
Word 42:	Temperature #1.	
Word 43:	Temperature #2.	
Word 44:	Temperature #3.	
Word 45:	Seed Lift.	
Word 46:	Crucible Lift.	
Word 47:	Seed Rotation.	

Word 48:	Crucible Rotation.	
Word 49:	Power Limit.	(Final Setpoints)
Word 50:	Debug Continuously Address #1.	
Words 51 - 52:	Debug Continuously Data #1 (4 bytes).	
Word 53:	Debug Continuously Address #2.	
Words 54 - 55:	Debug Continuously Data #2 (4 bytes).	
Word 56:	Debug Continuously Address #3.	
Words 57 - 58:	Debug Continuously Data #3 (4 bytes).	
Word 59:	Debug Continuously Address #4.	
Words 60 - 61:	Debug Continuously Data #4 (4 bytes).	
Word 62:	Diameter (Calculated Value).	
Word 63:	Spare.	
Word 64:	Debug Continuously Type Flags (compare Appendix 12.3, Debug Variable types:) TYPE(1) + 16*TYPE(2) + 256*TYPE(3) + 4096*TYPE(4)	

Comment Records:

Byte 1:	Always -1.
Bytes 2 - 6:	as in Data Records.
Bytes 7 - 128:	Comment input (122 ASCII characters; only the first 79 are displayed by SHODAT).

Appendix 13: Czochralski Growth Control System Messages

In addition to immediate responses to operator commands, the CGCS may issue messages to the console and to a Documentation output (if activated) which need not obviously be triggered by operator entries. For reasons of brevity, only the messages which are not generated by the Command Interpreter are listed below in alphabetical order. (The Command Interpreter responses are self-explanatory and always immediately related to an operator entry.) In general, messages beginning with "*****" have informational character only, whereas "#####" may indicate a genuine error condition. The latter messages are, in general, accompanied by a "beep". (Exceptions to this rule are the Disk, Input, Output, Printer, and System error messages which are tagged with asterisks. They are generated by the FORTRAN-iRMX-80 Interface Routines (compare chapters 5.2.1.6, 5.2.2.2, 5.2.2.3, and 5.2.3.10) and are displayed on the console only.)

***** All Conditional Macros cleared *****

An Unconditional CLEAR command (i.e., a CLEAR command without any parameter) was entered from the console or from a Macro Command file.

***** Automatic RESET executed - automatic Mode changes will follow *****

The operation mode was changed into a Diameter controlled mode while the Diameter Evaluation routines were not yet initialized with a RESET command. The system takes care of this situation on its own in a somewhat complicated procedure.

Can't calculate diameter with zero seed lift speed

The actual seed lift speed was still zero when RESET was commanded, or it is set to zero while the Diameter Evaluation routines are active.

Can't control system

The operator or a Macro Command attempted to SET or CHANGE a parameter or Variable while in Monitoring mode. The command is executed, though, but it may

become ineffective in the case of a change to any controlled mode.

Can't ramp parameter

The maximum number of parameters or Variables (20) were already being ramped when a SET or CHANGE command with non-zero transition time was issued. The change is effected immediately, without ramping.

Command Macro call ignored

A specified Macro was not found, or a disk error occurred while the Macro file header was read.

***** Command Macro preempted *****

A Macro Command was activated, either from a pending Conditional command, or through an unconditional Macro Command, while another Macro was active.

***** Conditional Macro cleared *****

A Selective CLEAR command has removed one Conditional Macro from the Conditional Command queue. This message is repeated for each Conditional Command cancelled with a Selective CLEAR; it may therefore appear multiplely.

Conditional Macro Command ignored

A Conditional Macro Command was encountered while already the maximum number of Macro Commands (8) were pending.

***** Conditional Macro started *****

The condition specified with a Conditional Macro command was found to be met; the Conditional Macro Command is activated.

Continued speed overflow - RESET required

The system cannot automatically recover from a serious problem.

Crystal shape adjusted

The calculated diameter value changed faster than permitted. The diameter value stored in memory for the diameter and crucible position evaluation is corrected to differ exactly by the permitted maximum from the value stored before. Crystal shape adjustments may cause minor transients in the calculated diameter and/or crucible position setpoint.

***** DISK ERROR xxx yy (TASK tsknam, LOC hexl) *****

Disk error message provided by the FORTRAN-iRMX-80 Interface routines (compare chapter 5.2.3.10 and Appendix 4).

***** End of Macro command file *****

The end of a Macro file was reached, or a disk error prohibited its further execution.

***** Executing Macro MACNAM *****

The Command file with the name MACNAM was started either from an unconditional Macro Command, or from a Conditional Macro Command whose condition was met.

Illegal command file format

A Macro Command file has an improper format and cannot be processed.

***** INPUT ERROR *****

Error message generated by the FORTRAN-iRMX-80 Interface routines, most likely due to illegal data entry on the console (compare chapter 5.2.2.2). This message should hardly appear, though.

Appendix 13: Czochralski Growth Control System Messages

Macro command not executable

A command referring to a Variable or absolute memory location was encountered in a Macro Command file generated for a different CGCS version. The command is ignored.

Macro MACNAM doesn't exist

The Macro Command with the name MACNAM was supposed to be executed either from an unconditional or from a Conditional Macro Command but the file MACNAM.CMD was not found on drive 0. The command is ignored.

Meltback detected

The crystal's length was reduced by more than approximately 1.2 mm since an earlier pass of the Diameter Evaluation routines. The Diameter Evaluation routines continue operating normally.

Mode automatically set to Manual

A zero seed lift speed or a speed overflow error was detected by the Diameter Evaluation routines.

***** New Mode: MODE NAME *****

The CGCS operation mode was set to the mode indicated, either from the operator console, from a Macro Command, or, automatically, in case of a diameter evaluation error.

Non-matching Command Macro system version - restricted command set

A Macro Command file generated under or for a different CGCS version was invoked. All commands referring to Variables or absolute hexadecimal addresses will be skipped.

***** OUTPUT ERROR *****

Error message generated by the FORTRAN-iRMX-80 Interface routines (compare chapter 5.2.2.3). This message should never be encountered!

Overflow - result limited to permitted maximum

As a result of a SET or CHANGE command, a parameter or Variable would have been set to a value exceeding the permitted range for the particular location.

Oxide height overflow - Diameter may be incorrect

The height of the boric oxide melt exceeded the permitted maximum of ca. 75 mm. The maximum melt height is used for diameter and crucible position setpoint evaluation. These data may therefore be incorrect.

Parameter can't be negative

A SET or CHANGE command attempted to set a diameter, temperature, or power limit setpoint to a negative value. The setpoint is set to zero instead.

***** PRINTER NOT READY *****

Error message generated by the FORTRAN-iRMX-80 Interface routines (compare chapter 5.2.2.3). The printer was in off-line mode while the system attempted to transfer data to it.

PROGRAM CODE DAMAGED AT xxxxH

At least one byte within the memory page (= 256 bytes) starting at the address specified in the message was changed since the last pass of the code checking routine, approximately 30 seconds ago. This message should never appear! Preserve all data of the run if it does happen, and report it immediately.

Appendix 13: Czochralski Growth Control System Messages

***** Recorder channel N is negative *****
***** Recorder channel N is positive *****

The output to the chart recorder channel N (N is an integer between 1 and 8) changed its sign. Initially, the output data of all channels is supposed to be positive.

***** Regular growth resumed *****

A meltback, zero seed lift speed, or speed overflow condition has been terminated; the Diameter Evaluation routines can resume correct operation.

Speed overflow

The calculated length of the crystal was increased or decreased by more than 2 mm during the last 10 seconds. This may be due to a very fast seed lift, or to an abrupt change of the crystal's weight. The system tries to recover automatically from such a condition.

***** SYSTEM ERROR (TASK tsknam, LOC hex1) *****

This error message is issued by the FORTRAN-iRMX-80 Interface routines (compare chapter 5.2.1.6); it should never appear. Preserve all data of the run if it does happen, and report it immediately.

Appendix 14: Dynamic Behavior of the PID Controller Routine

Simulations of the PID controller's response were performed for the most important operation modes in order to compare their dynamic response to various shapes of the error signal. For all simulations shown in the subsequent illustrations, the following parameters were used:

Proportional Multiplier	P = 256
Integral Multiplier	I = 64
Derivative Multiplier	D = 256
Limit	L = 25
Integral Scaling Factor	IS= 256
Bias	B = 0

The setpoint S was kept at 0, and the Actual signal A was set to follow the function depicted in Fig. A1.

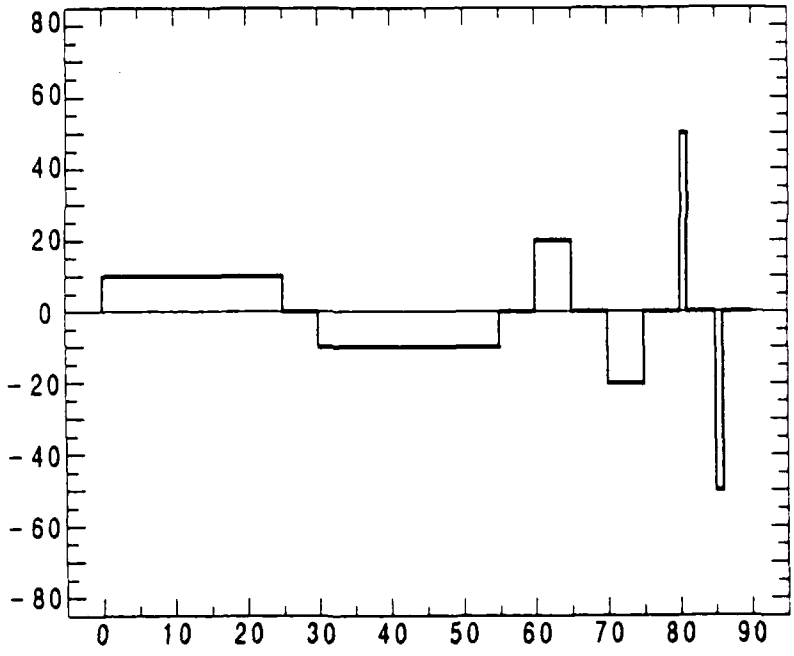


Fig. A1: "Actual" input signal used for the simulations.

The first part of this simulation, consisting of two series of 25 passes of FRPIDC with A equal to +10 and -10, respectively, was chosen to represent a small but persistent error for which the proportional (plus derivative) components of the output signal are well below a limit value (if one was chosen). Dur-

ing the next part of the simulation, A was increased to ±20 units for 5 passes each; for the ensuing error, the limit is to be incurred essentially due to the proportional and derivative components. The simulation is concluded with two single-pass pulses of A with a magnitude of ±50 units which were provided to represent the behavior of the controller for large transients.

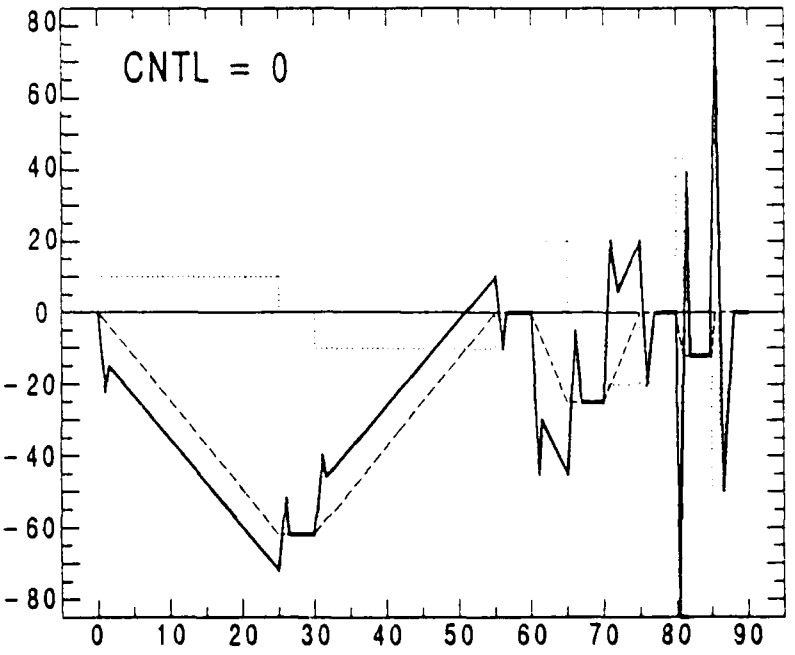


Fig. A2: Controller output signal (full line) and error integral (broken line) for unlimited operation with no option active.

In Fig. A2, the controller's response is shown for a CNTL value of 0, i.e., for no limiting and anti-windup operation. Note that, according to eq. (1) in chapter 5.3.2.1, the sign of the controller's output is opposite to the sign of the input value A. (This approach results in positive controller parameters for most applications.) During the first part of the simulation, the response of the controller is essentially determined by the integral component; the proportional and derivative components are only superimposed. Note that it takes a long time after the error reversed its sign until the controller's output signal (full line) changes its sign. The dynamic response is improved during part 2 of the simulation since the proportional and derivative components dominate

there. The concluding single error pulses result in a strong output signal in the proper direction, followed by the opposite overshoot caused by the reaction of the derivative component to the trailing edge of the pulse. Since the controller is linear and the input signal symmetrical, the error integral returns to zero after each part of the test.

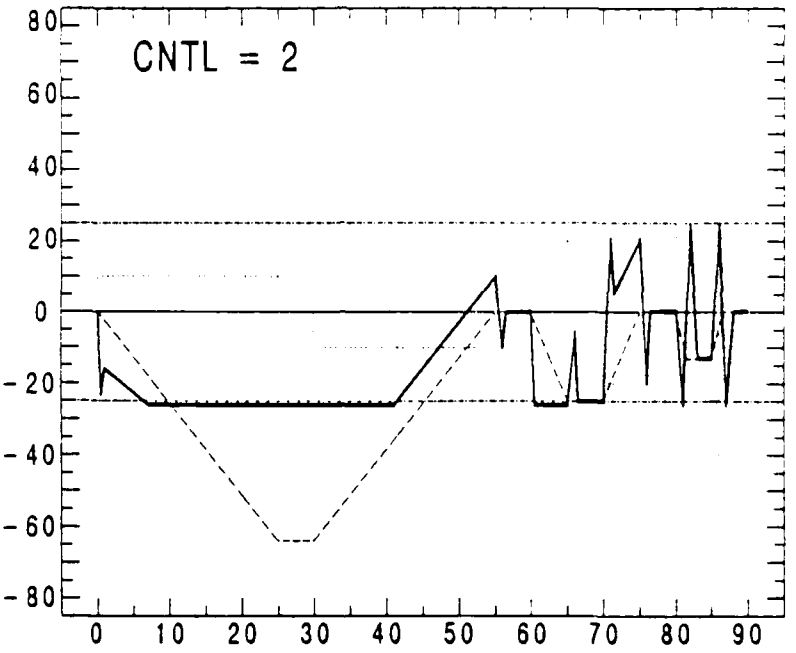


Fig. A3: Controller output signal (full line) and error integral (broken line) for output signal limiting with no anti-windup.

This is also true for the second set of control flags tested, namely, for CNTL equal to 2 (Fig. A3). In this case, the controller's output is limited to ±25, but aside from this limiting, the controller is still linear. The major drawback of simple output limiting can be seen in the first part of the controller's response curve, where there is no indication in the output signal that the error went to zero, and eventually changed its sign. (Note that, due to internal programming reasons, the output signal is limited to -26 units rather than -25. This fact is very unlikely to matter in actual applications, though, considering an output signal range of the controller of ±32767 units.) Similarly, the controller goes into saturation immediately at the beginning of the second part of the test, and remains there, although the error drops

back to zero, aside from a short spike caused by the derivative component. It requires a considerable error with the opposite sign to obtain an output signal with the expected direction. Output limiting also strongly affects the response to large transients: The controller's output bounces back and forth between its negative and positive limits. Since transients of this kind are most likely artifacts which better should not be regarded by the controller at all, output signal limiting obviously contributes to a suppression of these short pulses; the positive and negative spikes will cancel their effects mutually in most applications.

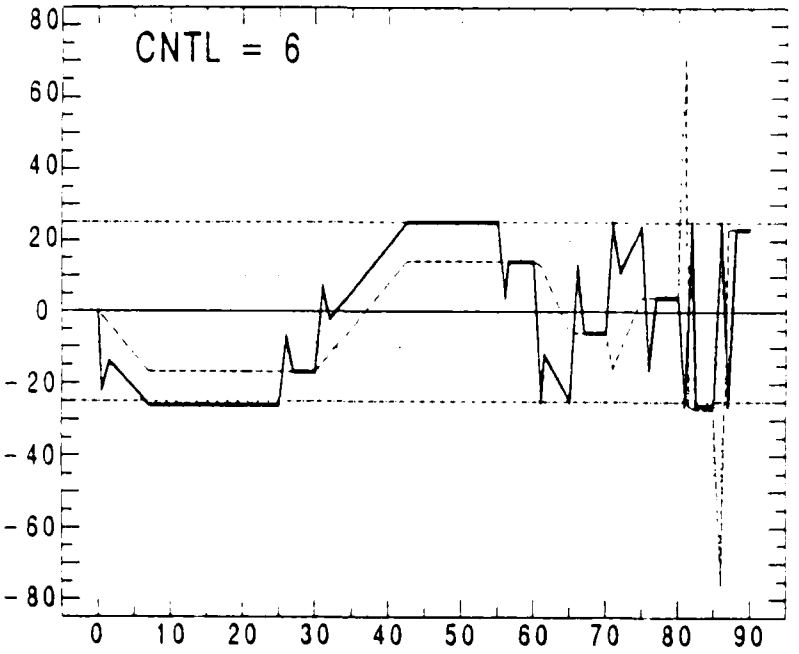


Fig. A4: Controller output signal (full line) and error integral (broken line) for output signal limiting with anti-windup mode A.

In order to improve the dynamic response, particularly, to long-term error conditions where the integral component predominates, the anti-windup function mode A was provided in FRPIDC. Fig. A4 shows the response of the controller with this feature activated in addition to output signal limiting (CNTL = 6). Indeed, the transition of A from 10 to 0 units has a clear influence upon the output signal, and a response with the expected sign is almost immediately obtained when A changes from 0 to -10 units. There is also a reasonable

response to the larger error pulses in the second part of the test; in fact, the response is very similar to the one obtained for no output limit, but, for anti-windup mode A, the output signal is better centered around zero independent from the preceding history of the controller. However, mode A fails totally for the large transients of the third part of the test. While there is only a small effect of a transient on the steady-state signal after the transient in the operation modes discussed above, mode A sets the error integral to a large value whose sign depends on the relative magnitude of the proportional and the derivative multipliers and the previous history of the controller; in fact, any output value between the positive and the negative maximum may ensue after a larger transient.

In order to counteract this not very desirable behavior, anti-windup mode B was developed where the error integral is not set to a value depending on the proportional and derivative components if the output exceeds the limit as it was in mode A; in contrast, the error integral is clamped to the positive or negative limit value, depending on the sign of the total controller output. The response in mode B (CNTL = 14) is shown in Fig. A5. There is no big difference between modes A and B for small long-term errors, although mode B reacts more slowly to changes of the controller's input than mode A does. The end of the first part of the test sequence, and, even more pronounced, the end of the second part shows, however, the main drawback of this mode: The error integral tends to "get stuck" at either of the controller's limits, and positive action (i.e., an input signal which eventually will reverse the error integral's sign) is required to remove it from there. Furthermore, some anomalies may also happen in mode B when large error transients are encountered. More or less the expected result is returned for the first spike: The output signal (and the error integral) bounces from the positive to the negative limit, and returns to the positive limit. The treatment of the second spike is less obvious. There is no visible response to the leading slope of the spike because the positive output which would have resulted from it is clipped off by the limiting operation. During the trailing slope, however, the derivative component determines the output signal. Incidentally, an output signal resulted in our simulations which was next to the negative limit but, from the controller's point of view, not beyond the limit. The integral component was therefore not modified but remained at its positive limit. (Had we used a pulse amplitude of 51 rather than 50 units, we would have obtained a reversal of the integral component's sign.)

Appendix 14: Dynamic Behavior of the PID Controller Routine

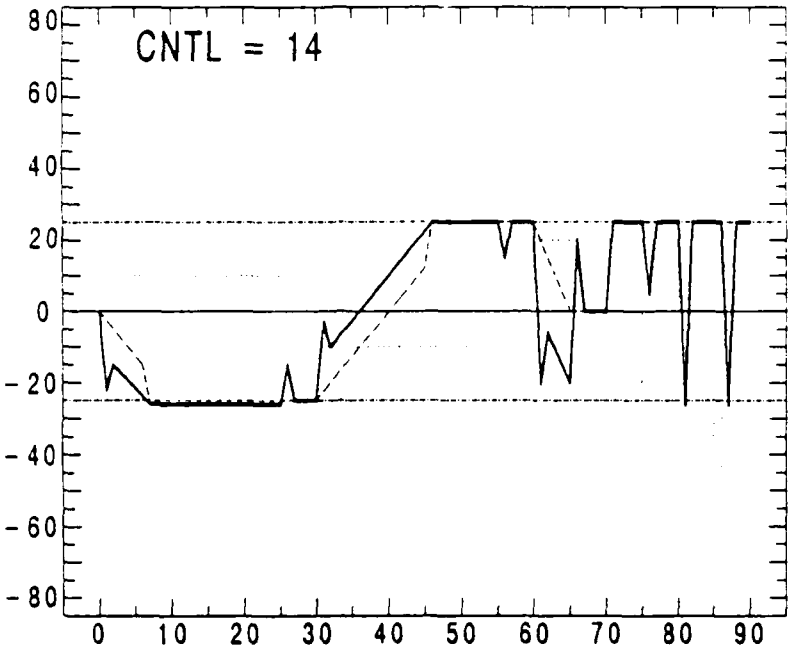


Fig. A5: Controller output signal (full line) and error integral (broken line) for output signal limiting with anti-windup mode B.

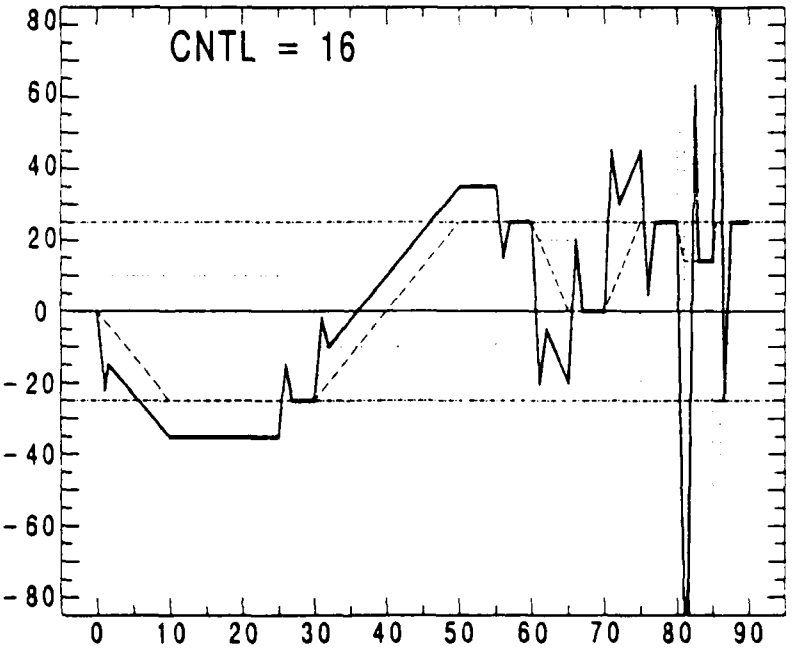


Fig. A6: Controller output signal (full line) and error integral (broken line) for integral limiting but no output signal limiting.

Appendix 14: Dynamic Behavior of the PID Controller Routine

The final two simulations (Figs. A6 and A7) were based upon a different approach: Rather than modifying the error integral when the output signal exceeds a limit, the error integral itself is kept within the bounds of a limit, no matter what the output signal looks like. This error integral limiting may be used with or without limiting of the final controller output. (The same limit value must be used, though, in both cases.) A simulation without output limiting (CNTL = 16) is shown in Fig. A6, while Fig. A7 shows the effects of output limiting (CNTL = 18). Again, the behavior of the controller suffers from the nonlinear response of the error integral which does not return to zero when an error condition occurred although the input signal is symmetric. (It is questionable, however, how representative the test signals chosen here are with respect to actual operating conditions.) Indeed, the response shown in Fig. A7 for integral and output limiting is very similar to the one obtained with anti-windup mode B in Fig. A5. The only differences occur for the handling of the large transients in the third part of the test. In this case, integral limiting seems to render more consistent and reasonable results, compared to the anti-windup schemes.

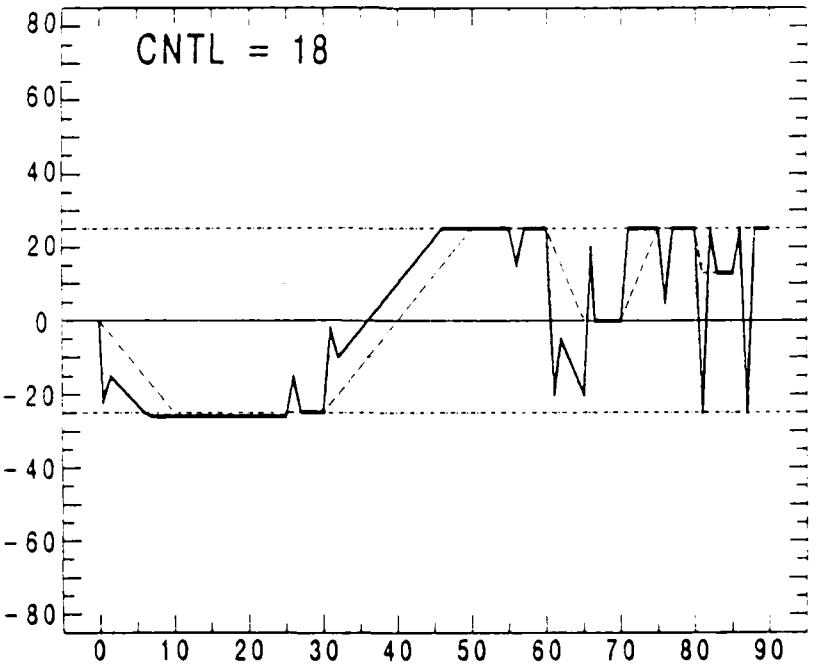


Fig. A7: Controller output signal (full line) and error integral (broken line) for integral and output signal limiting.

Appendix 14: Dynamic Behavior of the PID Controller Routine

Changing the integral scaling factor IS from 256 to 65536 has no effect on the behavior of the controller except that the integral reacts by a factor of 256 slower. Setting IS to 65536 and I to 16384 (64*256) resulted in exactly the same responses as discussed above.

Note that the PID controller routine checks for an output signal overflow after it calculated (and possibly limited) the error integral. In general, there is no point to keep integral limiting and any of the anti-windup schemes active at the same time because the anti-windup algorithms will override (and overwrite) the results of the integral limiter, except for some extremely weird operating conditions where an error reduces its magnitude at a rate fast enough to have the proportional component of the controller output overcompensated by the derivative component, without the error changing its sign. In this case, limiting of the error integral might occur without the output signal exceeding the limit. For obvious reasons, this case was not investigated; for all practical purposes, operation modes 20 through 23 and 28 through 31 are identical to the corresponding modes 4 through 7 and 12 through 15.